计算机 C/C++ 语言系列丛书

（OWL 2.0 for c++）

Borland C++ 4.0 for Windows

面向对象程序设计

李家圣 编著

希望

学苑出版社

95 年 Simon & Schuster 公司新到中、英文版电脑图书

序号	购书编号	书名	定价
1	1285	AutoCAD 12.0 循序渐进教程	39.00 元
2	1335	Windows NT 使用大全	77.00 元
3	1276	Windows 95 入门	18.00 元
4	1281	Windows 95 揭秘	39.00 元
5	1292	Visual Basic 趣味程序设计	31.00 元
6	1207	PC Tool 8 快速参考手册	13.00 元
7	1312	AutoCAD 12.0 参考手册	44.00 元
8	1221	AutoCAD 12.0 定制大全	73.00 元
9	1219	Windows NT 入门七钥	29.00 元
10	1236	FoxPro 2.6 for Windows 开发员指南(英文版)	138.00 元
11	1331	C++程序设计技术与实践	31.00 元
12	1319	UNIX 环境下的 C 语言程序设计	43.00 元
13	1286	Excel 5 快速入门	29.00 元
14	1180	Access for Windows 精粹	13.00 元
15	1368	Windows 95 简明指南	15.00 元
16	1371	Windows 趣味图形程序选	13.00 元
17	1333	调制解调器使用教程	15.00 元
18	1264	Internet 入门(信息高速公路)	29.00 元
19	1459	Adobe Photoshop 基础教程	39.00 元
20	1395	AutoCAD 培训教程	39.00 元
21	1445	轻松学习 PowerPoint 4	29.00 元
22	1421	Internet 用户手册(信息高速公路)	27.00 元
23		PowerPoint 4.0 for Windows 循序渐进教程	25.00 元
24		高级汇编语言	59.00 元
25	1391	精通 Visual Basic for Windows(3.0)	39.00 元
26	1483	PC 机汇编语言程序设计	45.00 元
27	1492	Windows 趣味声音软件包	15.00 元
28	1475	Turbo C++ Visual Edition for Windows 21 日通	35.00 元
29	1443	无代码数据库设计技术	34.00 元
30	1388	计算机程序员词典	39.00 元
31	1456	Borland C++程序员图形指南	59.00 元
32	1384	Inside CorelDraw 5 深入剖析(英文版)	136.00 元
33	1492	趣味声音软件包	15.00 元
34		轻松学习 Internet	29.00 元
35		轻松学习 MS Office	39.00 元
36	1478	DOS 6.0 开发者指南	59.00 元
37	1528	ADOBE ILLUSTRATOR 用户手册	69.00 元
38	1474	Access Basic 教程	17.00 元
39	1499	精通 MS Office(专业版)	89.00 元
40	1511	Internet 袖珍参考手册	19.00 元
41	1521	Windows 图形与动画	45.00 元
42		实用 DOS 6.2 指南	48.00 元
43		Word 6.0 for Windows 快速教程	35.00 元
44		Visual C++面向对象程序设计自学二十一日通	69.00 元
45		调制解调器使用指南	26.00 元
46		IBM PC 及其兼容机的通信与组网	46.00 元
47		用 Turbo C++编写游戏软件	49.00 元
48		组网技术入门指南	27.00 元
49	1501	扩展 Extending Visual Basic for Windows	29.00 元
50	1507	深入剖析 AutoCAD 13.0 for Windows/Windows NT	98.00 元
51		电子邮件包使用与范例	49.00 元
52	1425	IBM 多媒体使用手册	25.00 元
53	1363	电子邮件包的使用与范例	24.00 元
54	1462	自学 Visual Basic 3.0 for Windows 程序设计	49.00 元
55		自学 MSC/C++ 7.0 程序设计	59.00 元
56		MSC/C++ 7.0 开发员指南	99.00 元
57	1515	计算机图形环境	29.00 元
58		QBASIC 游戏大全	39.00 元
59		AutoLISP 12.0 编程大全	69.00 元
60	1523	开发 Windows 联机帮助系统	31.00 元
61		C/C++快速进阶教程	59.00 元
62		QBasic 轻松编程(教材)	19.00 元
63		Visual C++ 2.0 程序员必读	45.00 元

欲购以上图书的朋友请与 010－2541992，2562329 书刊部联系，

或传真 010－2579874，2561057 书刊部

ISBN 7-5077-0875-6

9 787507 708752 >

ISBN7－5077－0875－6/TP・ 24

本册定价：23.00 元

计算机C/C++语言系列丛书

Borland C++ 4.0 for Windows 面向对象程序设计 (OWL 2.0 for C++)

李家圣　编著
王　真　审校

学苑出版社

（京）新登字 151 号

内 容 提 要

本书主要讲述了如何编制 ObjectWindows 应用程序、ObjectWindows 动态链接库、Visual Basic 控制对象等方面的内容。在第五章有很多的应用实例，所有的程序均经过上机调试，以方便广大的编程爱好者。本书内容详尽，结构新颖，适合于从事编程或设计的各类人士参考。

需要本书的用户，请直接与北京海淀 8721 信箱书刊部联系，邮政编码 100080，电话 2562329。

计算机 C/C＋＋语言系列丛书

Borland C＋＋ 4.0 for Windows

面向对象程序设计（OWL 2.0 for C＋＋）

编　　著：李家圣

审　　校：王　真

责任编辑：甄国宪

出版发行：学苑出版社　　　**邮政编码**：100036

社　　址：北京市海淀区万寿路西街 11 号

印　　刷：列电印刷厂

开　　本：787×1092　1/16

印　　张：16.75　　　**字　　数**：390 千字

印　　数：1～5000

版　　次：1994 年 10 月北京第 1 版第 1 次

ISBN7－5077－0875－6/TP・24

本册定价：23.00元

学苑版图书印、装错误可随时退换

目　录

第一章　ObjectWindows 概述 …… 1
1.1　使用类等级的继承 …… 1
1.2　对象类型 …… 5
第二章　学习编制 ObjectWindows 应用程序 …… 10
2.1　应用程序对象 …… 10
2.2　窗口对象 …… 19
2.3　事件处理 …… 34
2.4　对话框对象 …… 41
2.5　控制对象 …… 57
2.6　界面对象 …… 78
2.7　菜单对象 …… 85
2.8　图形对象 …… 88
2.9　配件及配件窗口对象 …… 117
2.10　Doc/View 对象 …… 136
2.11　合法性检验器对象 …… 155
2.12　打印机对象 …… 160
2.13　总结 …… 165
第三章　ObjectWindows 动态链接库 …… 167
3.1　编写 DLL 函数 …… 167
3.2　编写共享的 ObjectWindows 类 …… 170
3.3　TModule 对象 …… 170
3.4　从非 ObjectWindows 应用程序中调用 ObjectWindows DLL …… 171
3.5　隐含加载和显式加载 …… 172
3.6　混合使用静态和动态链接库 …… 172
第四章　Visual Basic 控制对象 …… 173
4.1　使用 VBX 控制 …… 173
4.2　VBX 控制类 …… 174
4.3　处理 VBX 控制消息 …… 176
4.4　访问 VBX 控制 …… 178
第五章　应用举例 …… 181
5.1　第一步：设计一个简单的应用程序 …… 181
5.2　第二步：处理窗口事件 …… 182
5.3　第三步：在窗口中绘制文本 …… 187
5.4　第四步：在窗口中画线 …… 188
5.5　第五步：改变画线宽度 …… 191
5.6　第六步：重绘窗口和增加菜单 …… 196

5.7 第七步:应用通用对话框………………………………………………………………………… 203
5.8 第八步:在窗口中画多条线……………………………………………………………………… 210
5.9 第九步:修改画笔…………………………………………………………………………………… 214
5.10 第十步:增加装饰 ………………………………………………………………………………… 218
5.11 第十一步:转向 Doc/View 模块 ……………………………………………………………… 223
5.12 第十二步:讨论 MDI ……………………………………………………………………………… 238
5.13 第十三步:学习编写 DLL 程序………………………………………………………………… 254
5.14 第十四步:在 OWL 程序中使用自己编写的双链表 …………………………………… 257

第一章 ObjectWindows 概述

ObjectWindows 2.0 是为 Windows 3.1，Win32s 和 Windows NT 开发的应用程序框架。ObjectWindows 让你快速容易地开发具有 Windows 所有风格的应用程序。

ObjectWindows 2.0 具有以下特点：

- 容易在 16 位和 32 位平台间方便移植；
- 自动消息分发；
- 意外差错和错误处理；
- 允许移植到其他编译器和环境；
- 封装了 Windows GDI 对象；
- Doc/View 类使得数据抽取和显示更加简便；
- 打印机和打印预浏览类；
- 支持 Visual Basic 控制；
- 输入合法性检测器；

本章简单浏览一下 ObjectWindows 2.0 的等级结构和 ObjectWindows 2.0 的类库，解释每种类是如何与其他类相配合的，指导读者获取每个类的具体信息以便使用它们。

1.1 使用类等级的继承

这一部分讲述类的基本属性，重点放在 ObjectWindows 类上。包括三个主题：

- 如何处理类
- 继承的成员
- 成员函数的类型

1.1.1 使用类

对一个类，可以做三件最基本的事：

- 从它派生出一个新类
- 将它加到另外一个类中
- 对该类实例化（创建该类的一个实例）

一、派生出一个新类

给一个类增加或修改一些动作，可以从该类派生出一个新类，例如，我们从 TWindow 派生出类 TNewWindow：

```
class TNewWindow:public TWindow
```

```
{ public:
  TNewWindow(...);
  //...
};
```

派生一个新类时,可做三件事:

- 增加新数据成员;
- 增加新的函数成员;
- 重写继承来的成员函数;

增加新成员来增加或修改基类的功能;通过给派生类定义构造函数来调用基类的构造函数,并初始化增加的数据成员。

二、组合对象的行为

ObjectWindows 设计成使用多重继承。可以派生一个类继承多个类的行为。像单一继承一样,可以给多重继承的派生类增加成员函数和重写继承来的成员函数。

三、实例化类

要使用一个类,必须创建该类的实例。可以有几种方法实例化一个类:

- 使用标准的语法声明。就像声明标准变量 int 或者 char 一样。下列通过调用不带参数的 TMyApplication 构造函数来实例化 app 对象。

  ```
  TMyApplication app;
  ```

  这种声明方法只有在该类具有缺省构造函数或者构造函数的所有参数具有缺省值时才行。
- 也可用声明带有参数的构造函数实例化一个类。下列通过调用带有 char * 参数的 TMyApplication 构造函数来实例化 app 对象。

  ```
  TMyApplication app("AppName");
  ```

- 通过调用 new 操作符分配空间来实例化一个对象。如:

  ```
  TMyApplication * app;
  app=new TMyApplication;
  ```

- 也可以用 new 操作符,调用带有参数的构造函数实例化对象。如:

  TMyApplication * app=New TMyApplication("AppName");构造函数调用基类的构造函数来初始化数据成员。只能实例化非抽象类;即不含纯虚拟函数的类。

四、抽象类

抽象类具有纯虚拟函数,这个虚拟函数必须在派生类中重写。抽象类具有两个主要功用。一是创建其他类时提供一个概念上的框架;二是产生编码效果。

例如,ObjectWindows 类 THSlider 和 TVSlider 可以直接从 TScrollBar 派生。尽管一个

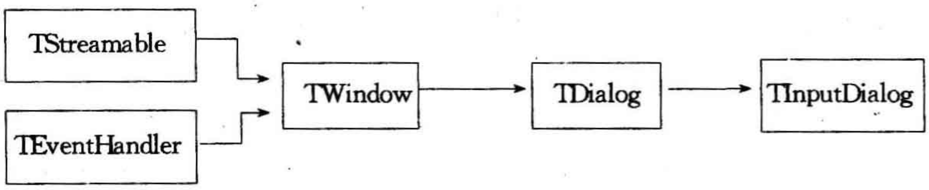
是垂直滚动块，另一个是水平滚动块，实际上它们具有相同的功能和响应。这种共性就保证可以创建一个抽象类 TSlider。THSlider 和 TVSlider 从 TSlider 派生，只须增加少量的特殊成员函数来画不同的滑块。

不可能实例化抽象类。其纯虚拟函数成员必须经过重写，以创建一个有用的实例。例如 TSlider 就不知道怎样画出自己或者响应鼠标事件。

如果要创建自己的滑块(例如图形滑块)，可能想从 TSlider 派生，但从 THSlider 或 TVSlider 派生可能更简单，这取决于哪个更符合要求。两种情况下，都要增加数据成员，增加或重载函数成员来增加所需的功能。如果想创建一个对角形滑块(西北－东南和西南－东北走向)，可能要创建一个中间抽象类 TAngledSlider。

1.1.2 继承的成员

下表显示了 TInputDialog 的继承关系：



TInputDialog 从 TDialog 继承而来，TDialog 又从 TWindow 继承，TWindow 则从 TStreamable 和 TEventHandler 继承而来。沿着继承等级路线可以增加更具体的动作。

下表列出了每个类的公共数据成员，包括从 TDialog 和 TWindow 基类继承来的成员：

TWindow	TDialog	TInputDialog
Status	Status	Status
HWindow	HWindow	HWindow
Title	Title	Title
Parent	Parent	Parent
Attr	Attr	Attr
DefautProc	DefaultProc	DefaultProc
Scroller	Scroller	Scroller
	IsModal	IsModal
		Prompt
		Buffer
		BufferSize

TInputDialog 继承了 TDialog 和 TWindow 的所有数据成员，并增加了一些数据成员。

要完全理解 TInputDialog，就必须理解它的继承关系：TInputDialog 既是一个对话框，又是一个窗口。TDialog 给窗口类增加模式的概念，TInputDialog 增加了存储和检索用户输入数据的能力。

1.1.3 成员函数的种类

有四种 ObjectWindows 成员函数：

- 虚拟函数
- 非虚拟函数
- 纯虚拟函数
- 缺省函数

一、虚拟函数

虚拟函数可在派生类中重载。虚拟函数和纯虚拟函数区别在于虚拟函数不一定非要被重载。虚拟函数提供了多态机制和一致的类接口能力,即使这些类功能大不相同。

二、非虚拟函数

不要重载非虚拟函数。因而,一个函数若要在派生类中重载,就应该将它定义成虚拟函数(事件响应表中的事件处理函数是一个例外)。例如,TWindow::CanClose 是虚拟函数,在派生类中可以重载它,以检测窗口是否可以关闭。而 TWindow::SetCaption 是非虚拟函数,因为不必要修改窗口标题设置的方式。

重载非虚拟函数导致的问题是:从派生类再派生的类可能试图使用重载函数。如果这个新的派生类不清楚父派生类已经改变了该派生的函数,这可能导致函数返回错误值,产生运行错误。

三、纯虚拟函数

在派生类中必须重载纯虚拟函数。通过下述方式将函数声明为纯虚拟函数。举例说明:TSlider::PaintRuler 是纯虚拟函数,声明是:

```
virtual void PaintRuler(TDC & dc)=0;
```

在派生类中,必须重载抽象基类中所有的纯虚拟函数。这样才能实例化派生类。绝大多数情况下,当使用标准的 ObjectWindows 类时,这样做不存在多大问题。纯大多数可以从其派生出类的 ObjectWindows 类不是抽象类。许多 ObjectWindows 类使用缺省的替代函数来代替纯虚拟函数。

四、替代函数

和纯虚拟函数不同,缺省的替代函数不用重载。这些函数提供最小的缺省动作或者干脆什么也不做。所以称为替代函数,是因为它们在程序中占有一个位置,在派生类中,可在这个位置放置代码。如 TWindows::EvLButtonDblClk:

```
inline void
TWindow:EvLButtonDblClk (UINT modkeys, TPoint &)
{ DefaultProcessing();
}
```

缺省时,EvLButtonDblClk 调用 DefaultProcessing 来响应双击鼠标事件。在自己定义的窗口类中,可以重载 EvLButtonDblClk 函数,将它定义在类的响应表中。定义的 EvLButtonDblClk 函数可以提供一些动作来处理用户双击鼠标的操作。也可调用基类的这个函数来

提供缺省处理动作。

1.2 对象类型

ObjectWindows 等级有很多不同的类，可以使用，修改或增加，可以将类分为以下几组：

- 窗口类
- 对话框类
- 控制类
- 图形类
- 打印类
- 模块和应用程序类
- Doc/View 类
- 各种 Windows 元素（杂项类）

1.2.1 窗口类

窗口是所有 Windows 应用程序重要组成部分。ObjectWindows 提供几种不同的窗口类：

- 窗口（一般窗口）
- 框架窗口
- MDI（多文档接口）窗口
- 修饰过的窗口

详细说明见后续章节。

一、窗口

TWindow 是所有窗口类的基类。它代表所有窗口的共同功能。不管这些窗口是对话框、控制还是 MDI 窗口。

二、框架窗口

框架窗口 TFrameWindow 从 TWindow 派生。它在 TWindow 上增加了框架窗口的功能，能拥有其他客户窗口。

三、*MDI* 窗口

多文档接口是管理多文档或一个应用程序里所有窗口的 Windows 标准。在 ObjectWindows 应用程序中，TMDIFrame，TMDIClient 和 TMDIChild 支持 MDI。

四、修饰窗口

TLayoutWindow 和 TLayoutMetrics 等几个类支持修饰控制（如工具条、状态条和消息条）。通过多重继承，可以在框架窗口和 MDI 框架窗口中增加修饰支持。这两个类是 TDecoratedFrame 和 TDecoratedMDIFrame。

1.2.2 对话框类

对话框 TDialog 从窗口类 TWindow 继承而来,用对话框类创建对话框,以处理各种各样的用户交互动作。典型的对话框包含控制来获得用户输入。后续章节将详细叙述对话框类。

一、公用对话框

除了应用程序创建的对话框外,ObjectWindows 支持 Windows 的公共对话框

- 选择文件(TFileOpenDialog,TFileSaveDialog);
- 选择字体(TChooseFontDialog);
- 选择颜色(TChooseColorDialog);
- 选择打印选项(TPrintDialog);
- 查找和替换文本(TFindDialog,TReplaceDialog)。

二、其他对话框

ObjectWindows 还提供其他一些对话框,这些对话框不是基于 Windows 公用对话框的。

- 输入文本(TInputDialog)
- 退出打印任务(TPrinterAbortDlg,常和 TPrinter 和 TPrintout 类一起用)。

1.2.3 控制类

控制类 TControl 从窗口类 TWindow 派生,以支持所有控制的通用行为。ObjectWindows 提供四种控制:

- 标准窗口控制
- 小工具(Widgets)
- 小配件(Gadgets)
- 修饰件(Decorations)

一、标准窗口控制

标准窗口控制有列表框、滚动条、按钮、复选框、互锁按钮、成组框、编辑控制、静态控制和组合框。

二、小工具

不同于标准窗口控制,ObjectWindows 小工具是用 C++写的特殊控制。ObjectWindows 提供的小工具有水平和垂直滚动条(THSlider 和 TVSlider)及量规(TGauge)。

三、小配件

小配件和标准窗口控制相似之处是:它也用来获取用户输入或给用户传送信息,但小配件和标准控制实现机理不同。不同于大多数界面元素,小配件不是窗口,不具备窗口句柄,不

能接收事件和消息，它不是基于 TWindow 的。

相反，配件必须位于配件窗口内，配件窗口负责配件的显示，所有的消息处理等，配件从配件窗口接收命令。

四、修饰件

修饰件是特殊的子窗口，用来让用户选择命令，给用户提供信息，有时还用来和用户交流特殊的信息。

- 控制条(TControlBar)让用户在窗口里安放一系列控制按钮，这些按钮常用作菜单的快捷键用。
- 工具框 TToo(Box)让用户在浮动板上安放一系列按钮。
- 消息条(TMessageBar)一般位于窗口的底部，用来向用户显示消息。例如 Borland C++集成环境用消息条显示菜单命令项的简单叙述。
- 状态条(TStatusBar)和消息条相似，不过同时可显示几条信息。例如在 Borland C++编辑窗口时，右下角显示 Insert 键，覆盖键(OverType)和一些错误信息。

1.2.4 图形类

Windows 提供功能强大但相当复杂的图形库叫 GDI。ObjectWindows 将 GDI 函数封装起来，这样使用设备上下文(DC)类和 GDI 对象(TGDIObject)时就不用太费神。

一、设备上下文类

通过 GDI 而不是直接在设备(如屏幕和打印机)上画，能够用设备上下文在位图里绘画。设备上下文是绘图工具，图形设置和有关图形设备及绘图状态等这些设备信息的集合。这样在使用 GDI 时，可以保证高度的设备无关性。下表是 ObjectWindows 将各种设备上下文封装后的图表：

设备上下文种类	ObjectWindows 设备上下文类
Memory	TMemoryDC
Metafile	TMetaFileDC
Bitmap	TDibDC
Printer	TPrintDC
Window	TWindowDC
Desktop	TDeskTopDC
Screen	TScreenDC
Client	TClientDC
Paint	TPaintDC

二、*GDI* 对象

TGDIObject 是其他几个类的基类，这几个类可以用来画图和控制图形显示。下表是 ObjectWindows 支持的这些类的列表：

GDI 对象	ObjectWindows GDI 类
Pens	TPen
Brushes	TBrush
Fonts	TFont
Palettes	TPalette
Bitmaps	TBitmap, TDib, TUIBitmap
Icons	TIcon
Cursors	TCursor
REgions	TRegion
Points	TPoint
Size	TSize
Rectangles	TRect
Color specifiers	TColor
RGB triple color	TRgbTriple
RGB quad color	TRgbQuad
Palette entries	TPaletteEntry
Metafile	TMetafilePict

1.2.5 打印类

由于封闭了和打印驱动程序通信的函数,TPrinter 类使得打印异常简单。TPrintout 将打印文档的任务封闭起来。第二章第十二节讲述如何使用打印类。

1.2.6 模块和应用程序类

Windows 应用程序负责初始化窗口,保证将 Windows 传递给它的消息送给合适的窗口。ObjectWindows 将这些行为封闭在类 TApplication 中。TModule 封闭了 DLI(动态连接库)行为。

1.2.7 文档/浏览器类

文档/浏览器类是通用文档——浏览器模块全部抽象而成的。TDocManager,TDocument 和 TView 是 Doc/View 的基类。Doc/View 模块是一个系统,数据包含在该系统中,由文档对象来存取这些数据。浏览器对象显示和操作这些数据。一个文档种类可以和任意数量的浏览器关联起来,用不同的方式来显示同一批数据。

例如,可以用两种方式显示一条线。一种是图形方式(在一个窗口中显示一条直线),一种是数字方式(将组成这条线的坐标点显示出来)。这样,就需要一个文档和两个浏览器:一个浏览器在屏幕上显示一条线,另一个浏览器显示组成直线的坐标点。也可以通过浏览器来修改数据。在上述情况下,就可以在图形显示器里画线来修改直线的数据,或者在数字显示器里输入数字修改直线的坐标点。

1.2.8 杂项类

一、菜单类

菜单可以是静态菜单，也可以修改甚至可以装入新的菜单。TMenu 及其派生类(TSystemMenu 和 TPopupMenu)用来操作菜单。

二、剪贴板

Windows 剪贴板是应用程序之间共享数据的主要方式。ObjectWindows 的 TClipboard 类使用户能在应用程序中支持剪贴板。

第二章　学习编制 ObjectWindows 应用程序

ObjectWindows 2.0 讲述了使用 ObjectWindows 应用程序框架编写 Windows 程序的很多基本知识。手册由十多个由简到繁的 ObjectWindows 应用程序组成。学习完这十多个步骤之后，读者就会对 Windows 应用程序的一些特点如菜单、对话框、图形控制条、状态条、MDI 窗口等，有一个全面的了解。本手册假设读者熟悉 C++编程并有 Windows 编程经验。开始编程之前，阅读第一章会有所帮助。这一章简单地浏览了 ObjectWindows 2.0 的类等级。

2.1　应用程序对象

ObjectWindows 2.0 用类 TApplication 和 TModule 分别封闭了窗口应用程序和 DLL 模式。类 TModule 对象封闭了窗口动态链接库(DLL)的初始化和关闭函数，类 TModule 还包含了 hInstance 和 LpCmdline 参数，它们等同于在非 ObjectWindows 应用程序中传递给 WinMain 函数的同名参数，注意函数 WinMain 和 LibMain 都有这两个公共的参数。类 TModule 更详细的讨论见有关章节。

类 TApplication 对象封装了 Windows 应用程序中的初始化，运行管理和关闭函数，类 TApplication 对象还包含了参数 hPrevInstance 和 nCmdshow 的值，它们等同于在非 ObjectWindows 应用程序中传给 WinMain 的同名参数，由于类 TApplication 是类 TModule 的派生类，所以类 TApplication 也具有类 TModule 所包含的功能属性。

另外，类 TApplication 对象包含容易调用的 Borland Custom Controls Library 库和 Microsoft Controls Library 库的一些函数，还有一个函数自动从象 Microsoft 3-D 那样的控制中获取标准控制的子类。详细的内容参见后续章节的有关部分 .

设计 ObjectWindows 2.0 应用程序不必明确地提供一个 WinMain 函数，这里可以用函数 OwlMain 代替。函数 OwlMain 用整型的(int)argc 和字符型的(char) * * argv 作为其参数，并返回一个整型值，就像传统的 C 语言和 C++语言设计的程序中用函数 main 一样。详细的内容见前面的叙述部分。

这一节讲述怎样使用类 TApplication 对象。设计者不必创建自己的 TModule 类对象，在动态链接库(DLL)下工作是个例外。在 ObjectWindows 应用程序中使用动态链接库(DLL)的更详细内容参见有关 DLL 的说明。

为了使用类 TApplication 对象，首先应完成以下几步：

- 包含正确的头文件
- 创建一个对象
- 得到这个对象

2.1.1　包含正确的头文件

类 TApplication 是在头文件 OWL\applicat.h 中定义的，所以在使用类 TApplication

时，应包含这个头文件，另外，由于类 TApplication 是类 TModule 的派生类，而头文件 OWL\applicat. h 包含头文件 OWL\module. h，所以程序中包含头文件 OWL\applicat. h 就可以了。

2.1.2 创建一个类对象

设计者可以用两个构造函数中的一个创建一个 TApplication 类对象，通常使用的构造函数是这样定义的。

```
TApplication (const char far * name);
```

类 TApplication 中此版本的构造函数有一个字符串参数，这个字符串就成了应用程序的名字，如果不指定一个名字，默认状态下，构造函数将字符串定义为空。类 TApplication 将这个字符串定义为应用程序的名字。

类 TApplication 的第二版的构造函数定义了一些参数，相当于通常情况下传给函数 WinMain 的参数：

```
TApplication (const char for * name, HINSTANCE instance,
              HINSTANCE prevInstance, const char far * cmdLine, int cmdShow);
```

设计者可以用这个构造函数将命令行参数传给类 TApplication 对象。

2.1.3

类 TApplication 包含了几个需要在应用程序对象外面调用的成员函数和数据成员，为了能访问它们，类 TWindow 有一个成员函数 GetApplication，它返回一个指向应用程序对象的指针，可以用这个指针去调用类 TApplication 的成员函数和访问类 TApplication 的数据成员。

下面列出了使用函数 GetApplication 的一种可能情况：

```
void TWindow::Error()
{
// display message box containing the application name
Message("An error occurred!",
        GetApplication() ->Name,MB_OK);
}
```

2.1.4 创建最小的应用程序

下面设计了一个最小的 ObjectWindows 应用程序：

```
#include<OWL\applicat. h>
int OwlMain(int/* argc */,char/*. * argv[]*/)
{
    return TApplication ("Minimum program"). Run();
}
```

这里创建了一个带有标题"Mininum Program"的主窗口的 Windows 应用程序，用户可以改变窗口尺寸、移动、最小化、最大化和关闭这个窗口。在实际应用中，将会从这个应用类

派生出一个具有更多功能属性的新类。

2.1.5 初始化应用程序对象

初始化一个 ObjectWindows 应用程序需要四步：

- 构造应用程序对象
- 初始化应用程序对象
- 初始化每个事例
- 初始化主窗口

一、构造应用程序对象

构造一个 TApplication 类对象时，首先调用成员函数 InitApplication，InitInstance 和 InitMainWindow 启动应用程序，可以通过重载这些成员函数来显示应用程序是如何初始化的。为了重载类 TApplication 中的函数，必须从类 TApplication 中派生出自已的应用对象类。

用在这里的类 TApplication 的构造函数将应用程序名作为其唯一的参数，它的默认值为 0，表示没有名，这个应用程序名用于默认的主窗口标题和错误信息中。应用程序类的构造函数应该调用类 TApplication 的构造函数。下面的例子是类 TApplication 的派生类的框架一部分：

```
#include<owl\applicat.h>
Class TMyApplication:public TApplication
{ public:
  TMyApplication(const char far * name=0)
  :TApplication(name{}
  ...
};
```

ObjectWindows 2.0 应用程序不要求一个显式调用 WinMain 函数，ObjectWindows 库提供了错误处理和意外事件处理机制。可以在函数 OwlMain 中执行任何想要的初始化工作。缺省的 WinMain 函数调用 OwlMain 函数。

为了构造一个应用程序对象，在函数 OwlMain 中创建一个应用类的实例。下面的范例显示了一个简单的应用对象类的定义，并创建了实例：

```
#include<owl\applicat.h>
class TMyApplication:public TApplication
{ public:
        TMyApplication(const char far *name=0)
        :TApplication(name){}
};
int OwlMain(int /*argc*/,char /* * argv[]*/)
{
    return TMyApplication("Wow!").Run();
```

```
}
```

(一)使用 **WinMain** 和 **OwlMain** 函数

在 ObjectWindows 2.0 中有一个默认的函数 WinMain,它提供了广泛的错误检查和意外事件处理功能,这个 WinMain 函数设置应用程序并调用 OwlMain 函数。

虽然可以在源文件中使用自己设计的 WinMain 函数,但似乎没有理由要这样做,任何情况下在函数 WinMain 中完成的工作都可以在函数 OwlMain 或者在类 TApplication 的初始化成员函数中完成。

下面的范例显示了在应用程序中函数 OwlMain 的典型应用:

```
#include<owl\applicat.h>
#include<string.h>
class TMyApplication:public TApplication
{ public :
        TMyApplication(char far  * name=0):TApplication(name){}
};
int OwlMain(int argc,char  * argv[])
{ char title[30];
    if (argc>=2)
        strcpy(title,argv[1]);
    else
        strcpy(title,"Wow!");
    return TMyApplication (title).Run();
}
```

如果经过考虑需要设计自己的 WinMain 函数,类 TAppliation 支持用另一个构造函数传递传统的 WinMain 函数的参数。下面的例子显示了怎样利用构造函数将函数 WinMain 的参数传给类 TApplication 对象:

```
#include<owl\applicat.h>
#include<string.h>
class TMyApplication:public TApplication
{ public:
        TMyApplication(const.char far  * name,
                                      HINSTANCE       instance,
                                      HINSTANCE       prevInstance,
                                      const char far  * cmdLine,
                                      int              cmdShow)
        :TApplication(name,Instance,prevInstance,cmdLine,cmdShow)
        {}
};
int PASCAL WinMain(HINSTANCE Instance,HINSTANCE prevInstance,
                   LPSTR lpszCmdLine,int nCmdShow)
```

```
{ return
  TMyApplication("MyApp",Instance,prevInstance,lpszCmdLine,nCmdShow).Run();
}
```

这个程序和 Borland C++3.1 程序非常相近，在 BC 3.1 版本中，TApplication 的构造函数见有关资料，读者可以对比一下。

二、初始化应用程序对象

用户可以同时多次运行同一个应用程序，从 16 位的应用程序看，第一个事例的初始化仅仅在另一个相同的应用程序不在运行时发生，每个事例的初始化每次都在用户运行应用程序时发生。如果用户启动又关闭一个应用程序，然后又启动，等等，则每个事例都是第一个事例，因为这些事例都不是同时运行的。如果在 32 位上运行，则每个应用程序运行在自己的地址空间上，而不共享事例数据，所以每个事例都是以第一个事例的身份出现，因此每次运行 32 位应用程序时，既完成第一个事例初始化又完成每个事例初始化。

如果当前的事例是第一个事例(通过将数据成员 hPreInstance 设置为 0 显示出来)，则函数 InitApplication 被调用。设计者可以在派生的应用程序类中重载函数 InitApplication，这个函数默认时不起作用。

例如，可以使用第一个事例初始化改变主窗口标题栏来反映当前的事例是不是第一个事例，为此需要在派生的应用程序类中增加一个名为 WindowTitle 的数据成员，在构造函数中，将 WindowTitle 设置为“Additional Instance”。重载函数 InitApplication 将 WindowTitle 置为“First Instance”。如果当前应用程序的事例是第一事例，调用函数 InitApplication，重写构造函数定义的主窗口标题名变量 WindowTitle 值。下面的例子列出了这段源程序代码：

```
#include<owl\applicat.h>
#include<owl\framewin.h>
#include<string.h>

class TTestApp:public Tapplication
{ public :
        TTestApp():TApplication("Instance Tester")
        { strcpy(WindowTitle,"Additional Instance");
        }
        protected:
          char WindowTitle[30];
          void InitApplication()
          { strcpy(WindowTitle,"First Instance");
          }
          void InitMainWindow()
          { SetMainWindow(new TFrameWindow(0,WindowTitle));
          }
};
int OwlMain (int argc,char * argv[])
{ return TTestApp().Run();
```

```
}
```

另外，应用程序在 32 位上运行时，可能并不像想像的那样工作，因为在 32 上位运行时，每个事例都把自己当作是应用程序的第一事例，同一应用程序同时多次运行将都在主窗口上显示标题"First Instance"。

三、初始化每个新的事例

用户可以同时多次运行同一应用程序，通过重载函数 TApplication::InitInstance 完成对每一事例所需的初始化。

函数 InitInstance 调用函数 InitMainWindow，然后创建并显示在函数 InitMainWindow 中定义的主窗口，如果重载函数 InitInstance，一定要保证新的 InitInstance 函数调用函数 TApplication::InitInstance。下面的例子显示如何利用函数 InitInstance 装入加速器表：

```
void
    TTestApp::InitInstance()
{
    TApplication::InitInstance()
    HAccTable=LoadAccelerators(MAKEINTRESOURCE(MYACCELS));
}
```

四、初始化主窗口

默认状态下，函数 TApplication::InitMainWindow 用与应用对象相同的名字创建一个框架窗口，这个窗口并不是很有用，因为它不能接受和处理任何输入的信息。必须重载函数 InitMainWindow 创建一个窗口对象来处理用户的输入，通常，函数 InitMainWindow 创建类 TFrameWindow 或 TFrameWindow 派生类的对象并调用函数 SetMainWindow。函数 SetMainWindow 有一个指针 TFrameWindow * 作为参数，并返回原来主窗口的指针（如果是一个没有设置主窗口的新的应用程序，则返回值为 0）。

下面例子中的应用程序创建了类 TFrameWindow 对象并将它作为主窗口：

```
#include<owl\applicat.h>
#include<owl\framewin.h>
class TMyApplication:public TApplication
{ public :
        TMyApplication():TApplication(){}
        void InitMainWindow();
};
void
TMyApplication::InitMainWindow()
{SetMainWindow(new TFrameWindow(0,"My First Main Window"));
}
int OwlMain(int argc,char * argv[])
{ return TMyApplication().Run();
```

}

运行这个程序时，标题栏上写的是"My First main Window"类 TApplication 的构造函数传递的应用程序名只有在没有提供主窗口时才有用。还是一样，这个例子不能完成太多的工作，因为仍然没有提供框架窗口来处理用户的输入，但是一旦派生出一个窗口类处理用户接口，就可以通过同样简单的办法显示这个窗口。

(一)指定主窗口显示模式

通过设置类 TApplication 中的数据成员是 nCmdShow，可以改变应用程序的主窗口，这个数据成员相当于函数 WinMain 中的参数 nCmdShow。一旦 Run 函数运行时便可设置这个变量值，直到调用函数 TApplication：InitInstance。这也显然意味着可以在函数 InitApplication 中或函数 InitMainWindow 中设置变量 nCmdShow 的值。

例如，如果你希望用户运行应用程序时，显示的窗口为最大尺寸，那么可以在函数 InitMainWindow 中设置 nCmdShow 的值：

```
#include<owl\applicat.h>
#include<owl\framewin.h>
class TMyApplication:public TApplication
{ public :
        TMyApplication(char for * name):TApplication(name){}
        void InitMainWindow()
        { SetMainWindow(new TFrameWindow(0,"Maximum Window"));
                nCmdShow=SW_SHOWMAXIMIZED;
        }
};
int OwlMain(int argc,char * argv[])
{ return TMyApplication("WOW").Run();
}
```

nCmdShow 作为函数 ShowWindow 或成员函数 TWindow::Show 的参数，可以设置为任意合适的值，像 SW_HIDE，SW_SHOWNORMAL 和 SW_NORMAL 等等。

(二)改变主窗口

在应用程序运行过程中，可以用函数 SetMainWindow 来改变主窗口。函数 SetMainWindow 有一个参数是 TFrameWindow *，并返回一个指向原来主窗口的指针(如果这是一个新的应用程序，还没有设置主窗口，则返回值是 0)。可以用这个指针保存原来的主窗口以便保存，这样可以有准备地用这个指针删除原来的主窗口对象。

2.1.6 应用程序的消息处理

一旦应用程序初始化后，应用程序对象的函数 MessageLoop 便开始运行，函数 MessageLoop 处理从 Windows 得来的所有消息，有两种方法可以简化从一个 ObjectWindows 应用程序中传来的消息。

■ 通过重载默认的消息处理函数,作额外的消息处理。
■ 挂起处理进程

一、额外的消息处理

类 TApplication 有一些成员函数为所有的 ObjectWindows 应用程序提供了消息处理功能。这些函数是 MessageLoop ,IdleAction,PreProcessMenu 和 ProcessAppMsg。

二、空闲进程处理

空闲处理是让应用程序利用没有消息等待的空闲时间(包括用户输入),如果没有消息等待处理,函数 MessageLoop 调用函数 IdleAction。

为了实现空闲进程,重载函数 IdleAction 实现真正的空闲进程。记住空闲进程是在用户不做任何事情时发生。因此空闲进程应该持续很短的时间,如果需要几十秒才能完成某一事情,则最好将其划分为几个进程。

函数 IdleAction 的参数(IdleCount)是 long 型的数值,函数 IdleAction 是在消息之间调用,可以用 IdleCount 参数在低优先权和高优先权之间选择,如果 IdleCount 达到一个较高的值,则有较长时间没有用户输入,所以较安全的是实现低优先权空闲处理。从函数 IdleAction 返回 TRUE,以便后来回头调用,应当经常调用基类中函数以便执行自己的过程。如果你正在编写 Windows NT 程序,你也可以使用多线程作背景过程。

2.1.7 关闭应用程序

用户通常选择菜单 File/Exit 或者按下 Alt+F4 键来退出 Windows 实用程序。这很重要,程序截获这个工作,并给用户机会存储未保存的文件。类 TApplication 能做到这一点。

一、改变关闭行为

类 TApplication 和其他所有类都具有或继承成员函数 CanClose,当应用程序试图关掉窗口时,它询问主窗口和文本管理窗口的 CanClose 函数,如果它们当中还有子窗口,则调用每个子窗口的 CanClose 函数,结果每个子窗口都依次调用自已的子窗口,如此类推。函数 CanClose 给每个对象一个准备关闭窗口的机会,如果必要,也提供给每个对象忽略关闭窗口的机会,如果已完成清理过程,则其 CanClose 函数返回一个真值 TRUE。

如果任何被调用的 CanClose 函数返回 FALSE,则忽略关闭操作。

二、关闭应用程序

函数 CanClose 系统给应用程序对象、主窗口、和其他任何窗口一个机会,或者为关闭窗口做准备,或者防止窗口被关闭。最后,应用程序对象完成关闭应用程序的过程。正常的关闭顺序是这样的:

(1) Windows 发出一个 WM_CLOSE 消息到应用程序主窗口。
(2) 主窗口对象的成员函数 EvClose 通过调用应用程序的成员函数 CanClose 予以响应。
(3) 应用程序对象的成员函数 CanClose 调用主窗口对象的成员函数 CanClose。

(4) 主窗口和文本管理器调用它们各自子窗口的成员函数 CanClose，只有当所有的子窗口的成员函数 CanClose 都返回真值时，主窗口和文本管理器对象的成员函数 CanClose 才返回真值 TRUE。

(5) 如果主窗口和文本管理器对象的 CanClose 函数都返回 TRUE，则应用程序对象的成员函数 CanClose 也返回真值。

(6) 如果应用程序对象的 CanClose 函数返回真值，则函数 EvClose 关闭主窗口并结束应用程序。

二、重新定义 *CanClose* 函数

CanClose 函数应当很少返回 FALSE，它应该完成返回 TRUE 的必需操作。CanClose 函数只有在不能顺序地完成关闭操作或用户需要应用程序继续运行时才返回 FALSE。

例如，一个编辑器窗口在关闭之前，它的 CanClose 成员函数应该检查其中文本是否修改过，然后提示用户是否要保存其中文本，一个具有 Yes，No 和 CanCel 的消息框是最合适的，如果用户选择了 Cancel，则表示用户不想关掉这个应用程序，所以 CanClose 函数返回 FALSE。

2.1.8　运用控制库

类 TApplication 有一些函数用于调用 Borland Custom Controls Library (BWCC. DLL 用于调用 16 位的应用程序，而 BWCC32. DLL 用于调用 32 位的应用程序)和 Microsoft 3-D Controls Library (CTL3. DLL)。这些动态链接库(DLL)普遍地给应用程序提供标准的界面对象。

一、使用 *Borland Custom Controls Library* 库

可以用函数 TApplication::EnableBWCC 打开或关闭 Borland Custom Controls Library，函数 EnableBCC 只有一个 BOOL 类型的参数，它返回一个 void 值，当把 TRUE 值传给函数 EnableBWCC 时，如果动态链接库没有装入则装入；当把 FALSE 值传给函数 EnableBWCC 时，如果动态链接库还没有释放则释放掉。

读者可以证实 Borland Custom Control Library 的动态链接库是否是通过调用函数 TApplication::BWCCEnabled 装入的。函数 BWCCEnabled 没有参数，如果 DLL 已经装入则返回 TRUE，否则函数 BWCCEnabled 返回 FALSE。其原理参见动态链接库的有关部分。

一旦装入了 DLL，就可以使用 Borland Custom Controls Library 的有规则的功能属性，函数 EnableBWCC 自动并且准确地打开库，而不考虑是 16 位应用程序或 32 位应用程序。

二、使用 *Microsoft 3-D Controls Library* 库

可以用函数 TApplication::EnableCtl3d 来装入或和释放 Microsoft 3-D Controls Library，函数 EnableCtl3d 有一个 BOOL 类型的参数，并且返回 void 类型的值。当传给函数 EnableCtl3d 的值是 TRUE，而且 DLL 还没有装入的话则装入它；反之，当传给函数 EnableCtl3d 的值是 FALSE，而且 DLL 还没有释放则释放它。

读者会发现，是通过调用函数 TApplication::Ctl3dEnabled 来装入 Microsoft 3-D Con-

trols Library 动态链接库的，函数 Ctl3dEnabled 没有参数，如果 DLL 被装入则函数返回 TRUE，否则返回 FALSE。

为了使用函数 EnableCtl3dAutosubclain，用函数 EnableCtl3d 装入 Microsoft 3-d Control Library 动态链接库。函数 EnableCtl3d Autosubclass 有一个 BOOL 类型的参数，并且返回一个 void 类型的值，如果将 TRUE 值传给函数 EnableCtl3dAutosubclass，则可自动生成子类；如果将 FALSE 值传给函数 Enable Ctl3d Autosubclass，则不可自动生成子类。

当能生成子类时，任何创建的非 ObjectWindows 对话框都有三维的效果，可以在创建对话框后立即关闭子类生成功能；没有必要在显示对话框时还让它起作用。

2.2 窗口对象

窗口对象是高层次的界面对象，能方便地创建、控制和处理窗口及其子窗口。

ObjectWindows 提供了以下几种窗口对象：

- 布置类窗口 TLayoutWindow
- 框架类窗口 TFrameWindow 及其派生类窗口。
- 修饰过的框架窗口 TDecoratedFrame
- MDI 窗口

以上几种窗口对象后续部分均有描述。另一种窗口对象类，叫作配件窗口 TGadgetWindow，后续部分将作详细说明。

2.2.1 使用窗口对象

这一节讲述如何创建、显示和填充窗口对象。它描述如何执行下列任务：

- 构造窗口对象
- 设置初始属性
- 创建窗口界面元素

这一节讨论几种不同类型的窗口——框架窗口、布置窗口、修饰框架窗口和 MDI 窗口它们都是窗口对象的实例，这部分的信息应用于所有这些不同类型的窗口对象中。

一、构造窗口对象

窗口对象描述了界面元素，对象是通过一个句柄元素相联系的，这个句柄元素存储在对象的数据成员 HWindow 中。HWindow 是从类 TWindow 中继承下来的，当构造一个窗口对象时，其界面元素并不存在，必须在单独的步骤中创建它。类 TWindow 还有一个构造函数可以用在动态链接库(DLL)中，为已存在的界面元素创建一个窗口对象。

在 ObjectWindows 2.0 中，有几个类把类 TWindow 和类 TFrameWindow 当作虚拟基本类，这些类是 TDialog、TMDIFrame、TTinyCaption、TMDIChild、TDecoratedFrame、TLayoutWindow、TClipboardViewer、TKeyboardModeTracker 和类 TFrameWindow。在 C++中，首先构造虚拟基本类，这就意味着，派生类的构造函数不能为基本类的构造函数定义默认的参数。解决这个问题有两种办法：

■ 在构造派生类时,明确地构造直接基本类和所有的虚拟基本类。
■ 使用虚拟基本类的默认构造函数。类 TWindow 和 TFrame Window 都有一个默认的构造函数,它们都有函数 Init,可以为基本类定义参数,在派生类的构造函数中调用函数 Init,可以为基本类定义任何需要的参数。

下面的例子显示了如何用上面描述的方法构造一个窗口对象:

1. TWindow * SampleWindow=new TWindow(this, "Window title")创建一个 TWindow 对象的指针,该指针指向的窗口是一个子窗口,其父窗口由 this 指针指定;
2. TFrameWindow SampleWindow(0,"Window title",new TMDIClient,TRUE);
   创建一个 TFrameWindow 对象,该对象作为主框架窗口。
3. class TSampleWindow:public TFrameWindow

```
{ public :
        TSampleWindow (TWindow * parent,char * title):TFrame Window(parent;title);
        TWindow(parent,title){ }
};
TWindow * childwin=new TSampleWindow(GetMainWindow()),"child Window");
```

   从 TFrameWindow 派生新类 TSampleWindow 注意 TSampleWindow 的构造函数,该构造函数依次调用基类 TFrameWindow,TWindow 的构造函数。从 OWL 的类等级可以看出这样做的理由,在实例化 TSampleWindow 时,将指向子类 TSampleWindow 的指针改为基类 TWindow 的指针,这在 C++编程中是很常见的。
4. class TNewWindow:virtual public TWindow

```
{ public :
        TNewWindow(TWindow * parent,char * title);
};
TNewWindow::TNewWindow(TWindow * parent,char * title)
{ Init (parent,title,IDL_DEFAULT);
}
TNewWindow * newwin=new TNewWindow(GetMainWindow(),"title");
```

以 TWindow 类作为虚拟派生出一个新类,在其构造函数中调用 Init 函数,该函数为虚拟基类的构造函数传递参数。

二、重载生成属性

一个典型的 Windows 应用程序拥有多种不同的类型的窗口:覆盖式的或弹出式的、加边与不加边的、可滚动的与不可滚动的、带标题的或不带标题的等等。这些不同的类型用参数,属性选择。通常这些类型属性以及窗口的标题等都是窗口对象生成之时设置的,而在界面元素生成之时才使用它们。

窗口对象的生成属性,像类型和标题等存储在对象的数据成员 Attr 中,它被定义为 TWindowAttr 类型。表 2.1 列出了 TWindowAttr 的数据成员:

表 2.1 窗口的生成属性

成 员	类 型	描 述
Style	DWORD	包含组合类型常量
ExStyle	DWORD	包含扩展类型常量
X	int	表示起始窗口水平位置，它是窗口左上角的水平屏幕坐标
Y	int	表示起始窗口垂直位置，它是窗口右上角的垂直屏幕坐标
H	int	表示窗口的起始高度，采用屏幕坐标
Menu	TResld	表示窗口的菜单资源 ID，不必直接地在函数 Attr. Menu 中定义菜单标识符，可以使用函数 AssignMenu 取代
Id	int	子窗口的 ID 用于在一个控制和其父窗口之间通讯，同一父类的所有子类 ID 必需是唯一的如果控制是在一个资源中定义，则它必须与资源保持一致，任何一个窗口不得同时定义 Menu 和 Id
Param	char far	用在 TMDIClient 中，保存关于 MDI 框架和子窗口的信息
AccelTable	TResld	窗口加速表资源的 ID

(一)重载默认属性

表 2.2 列出了窗口的默认生成属性。通过修改结构 Attr 中的值，来重载派生类以构造函数中的默认值。例如：

```
TTestWindow::TTestWindow(TWindow * parent,const char title)
:TFrameWindow(parent,title),TWindow(parent,title)
{
    Attr.Style &=(WS_SYSMENU|WS_MAXIMIZEBOX);
    Attr.Style|=WS_MINIMIZEBOX;
    Attr.X=100;
    Attr.Y=100;
    Attr.W=415
    Attr.H=355;
}
```

(二)子窗口的属性

可以在子窗口的构造函数中或在生成子窗口的的代码中，设置子窗口的属性。当改变父窗口对象的构造函数中的属性时，必须使用一个指向子窗口对象的指针，访问其数据成员 Attr 的值。

```
TTestWindow::TTestWindow(TWindow * parent,const char title)
:TFrameWindow(parent,title),TWindow(parent,title)
{ TWindow helpWindow(this, "Help System");
    helpWindow.Attr.Style |=WS_POPUPWINDOW|WS_CAPTION;
    helpWindow.Attr.X=100;
    helpWindow.Attr.Y=100;
    helpWindow.Attr.W=300;
```

```
        helpWindow.Attr.H=300;
        helpWindow.SetCursor(0,IDC_HAND);
    }
```

下面的表显示了可能重载的数据成员 Attr 的值：

表 2.2　默认的窗口属性

Attr 成员	默认值
Style	WS_CHIUD|WS_UISIBLE
ExStyle	0
X	0
Y	0
W	0
H	0
Menu	0
Id	0
Param	0
AccelTable	0

注意:0 表示使用 *Windows* 的默认值

三、生成窗口的界面元素

一旦构造了一个窗口的对象，就必须通知 Windows 创建相关的界面元素，通过调用对象的成员函数 Create 做到这一点：

```
Window.Create();
```

函数 Create 完成以下工作：

- 生成界面元素
- 将 HWindow 设置给界面元素的句柄
- 将结构 Attr 中的成员设置成实际的值(像 Style,ExStyle,X,Y,H,W)。
- 调用函数 SetupWindow

在 C++中，创建窗口对象的界面元素时，可能有两个意外，在应用程序可能碰到内存和资源问题时，应该停止调用 Create 函数，而去处理这些问题。当窗口不能生成时，Create 函数发出一个 TXInvalidWindow 意外操作；窗口中的子窗口不能生成时，函数 SetupWindow 发出 TxIvalidChildWindow 的操作。这两种意外通常都是由于内存不够或其他资源方面的原因引起。

应用程序的主窗口是通过 TApplication::InitInstance 自动生成的，通常不必调用 Create 创建主窗口。更详细的信息参见前面关于主窗口的内容。

2.2.2　布置类窗口

这部分讨论布置类窗口。布置类窗口封装在类 TLayoutWindow 中，它是从 TWindow

类派生来的。连同 TFrameWindow 和 TLayoutWindow 类一起，作为修饰框架窗口的基础，并提供在框架区域安排修饰的功能。

布置类窗口所以如此命名是因为它们能在布置类窗口的客户区域输出子窗口，子窗口的位置是根据布置窗口和另一子窗口来确定的。(另一子窗口被称为同胞子窗口)。子窗口的位置依赖于窗口的成员 layoutmetric，它包含了许多规则，这些规则描述了窗口的 X 坐和 Y 坐标，还有其高度和宽度等，这些规则通常根据同胞子窗口的坐标来定，当然，最终同布置类窗口的尺寸和安排有关。

子窗口的 layoutmetric 包含在名叫 TLayoutMetric 的类中，一个布置对象包含许多布置约束 Layoutconstraints。每个布置约束描述了一条寻找一个特定尺寸的规则，像 X 坐标，或窗口的宽度。完整地描述一个布置约束对象需要四个布置约束，布置约束包含在名叫 TLayoutconstraints 的结构中；但是，通常只使用 TLayoutconstaints 的一个派生类，像 TEdgeconstraint，TEdgeOrWidthconstraint 或 TEdgeOrHeightconstaint。

一、布置类约束

布置类约束指定了一个窗口的一条边和一个尺寸同其同胞窗口或父类布置窗口的一条边和一个尺寸的关系，这些关系可以是很弹性的，例如，可以将一个窗口的宽度设置成其父窗口宽度的百分比，所以一旦父窗口改变尺寸，子窗口在相关的区域同样改变尺寸；还可以将一个窗口的左边设置成另一子窗口的右边，这样当一个窗口移动时，另一窗口则被连在一起移动；还可以设置一个窗口的约束占用客户区域的绝对尺寸和位置。

通常使用的三种类型的约束是 TEdge Constraint，TEdgeOrWidthconstraint 和 TEdgeOrHeightconstraint，这些结构组成了类 TLayoutMetrics 中使用的所有约束，TEdgeconstraint 和 TEdgeOrHeightconstraint 是从 TEdgeconstraint 派生来的，从外面看，这三个对象几乎相同。这一部分在讨论 TEdgeconstraint 时，它是指所有这三个对象——TEdgeconstraint，TEdgeOrWidthconstraint 和 TEdgeOrHeightconstraint ——除非其余两个类明确排除在外。

(一)定义约束

最基本的定义约束关系(也就是指，设定一个窗口的边和尺寸同另一窗口的边和尺寸的关系)的方法是使用函数 Set，函数 Set 在类 TEdgeconstraint 中定义，相继地被类 TEdgeOrWidthconstraint 和类 TEdgeOrHeightconstraint 继承。

这里是函数 Set 的声明：

```
void Set(TEdge edge,TRelationship rel,
         TWindow * otherWin,TEdge otherEdge,
         int value=0);
```

其中：

■ edge 指定窗口的哪一部分被约束，为此，有一个 enum 类型 TEdge，它包含五个可能的值：

- ImLeft 指定窗口的左边。
- ImTop 指定窗口的顶边。

- ImRight 指定窗口的右边。
- ImBottom 指定窗口的底边。
- ImCenter 指定窗口的中间位置。这个对象拥有像 TLayoutMetrics 这样的约束,决定是垂直中心还是水平中心。

还可以将窗口的宽度或高度定义为一个约束,但这只适用于 TEdgeOrWidthconstraint 和 TEdgeOrHeightconstraint,为此有一个 enum 类型变量 TWidthHeight. TWidthHeight 有两种可能的值:

- ImWidth 指定窗口应该约束的宽度。
- ImHeight 指定窗口应该约束的高度。

■ rel 定义了两条边之间的关系:

rel	关 系
ImAsis	尺寸被当前值约束
ImPercentf	一个尺寸约束边尺寸的百分比约束,它通常用于一个约束的宽度或高度。
ImAbove	尺寸由约束边上的某一距离约束。
ImleftOf	尺寸由约束边的左边某一距离约束。
ImBelow	尺寸由约束边的底部的某一距离约束。
ImRightOf	尺寸由约束边的右边的某一距离约束。
ImSameAs	尺寸由与约束边相同的尺寸约束。
ImAbsolute	尺寸由绝对尺寸或坐标约束。

■ OtherWin 定义了约束子窗口的窗口,当指定父窗口时,必须使用值 ImParent。

■ OtherEdge 指定了窗口 OtherWin 的特定边,它用于约束子窗口,otherEdge 可以具有任何 edge 允许的相同值。

■ Value 依据 rel 的值意味几个可能的情况:

rel	Meaning of value
ImAsis	Value 无意义,必须设置为 0。
ImPercentOf	Value 显示了被约束尺寸应具有的约束尺寸的百分比。
ImAbove	Value 表示被约束边应具有的约束边上的单元数。
ImLeftOf	Value 表示被约束边应具有的约束边左边的单元数。
ImBelow	Value 表示被约束边应具有的约束边下边的单元数。
ImRight Of	Value 表示被约束边应具有的约束边右边的单元数。
ImSameAs	Value 无意义,必须设置为 0。
ImAbsolute	Value 表示约束边的绝对尺寸。 当 edge 取 Imleft,ImRight 或有时取 ImCenter 时 Value 是边的 X 坐标; 当 edge 取 ImTop,ImBottom 或有时取 ImCenter 时,Value 是边的 Y 坐标。 当 edge 取 ImWidth 或 ImHeight 时,边代表约束的尺寸。 对象本身决定着 ImCenter 表示着 X 坐标还是 Y 坐标。

value 的含义还依赖于 Units 的值，Units 是 TLayoutconstraint 中的一个 TMeasurement 成员，TMeasurementUnits 是一个 enum 类型，描述了 value 所代表的单位类型，Units 可以是 ImPixels 或 ImLayoutUnits。ImPixel 表示 value 是代表物理点的绝对数目，ImlayoutUnits 则表示 value 是代表逻辑单元的数目。这些布置单元由当前的布置窗口尺寸决定。

TEdgeConstraint 还包含了一些函数，用它们可以设置预定关系，这些关系可以在函数 Set 中指明。事实上，这些函数调用 Set 去定义预定义关系。可以用这些函数设定大多数约束关系。

下面四个函数工作方式相似：

```
void  LeftOf(TWindow * sibling,int margin=0);
void  RightOf(TWindow * sibling,int margin=0);
void  AboveOf(TWindow * sibling,int margin=0);
void  Below(TWindow * sibling,int margin=0);
```

这些函数的每一个都根据同胞(sibling)约束窗口，将子窗口按某种关系放置。窗口的边是预定义的，定义时，被约束的边设置时与函数名相反，而约束边则与函数名相同。

例如，函数 LeftOf 将子窗口设置在同胞窗口的左边，这就意味着子窗口的被约束边是 ImRight 而约束边是同胞窗口(sibling)的 Imleft。

可以用函数 Absolute 将子窗口设置成一个绝对的值：

```
void  Absolute(TEdge edge,int Value);
```

变量 edge 表示希望约束来哪一条边，value 取在 Set 函数中使用关系 ImAbsolute 的值。

有两个较短的函数可以使用：

```
void  SameAs(TWindow * otherWin,TEDge edge);
void  Percent Of(TWindow * OtherWin,TEdge edge,int percent);
```

这两个函数使用被约束窗口和约束窗口的同一条边。这就是说，如果指定 edge 取 Imleft，则子窗口的左边被约束为 otherWin 的左边。

(二)定义约束关系

一个单个的布置约束还不是以输出一个窗口。例如，指定一个窗口必须在另一窗口下面十个像素点，而不告诉窗口的宽度和高度，还有左边界或边界的位置，或底边界的位置，它只告诉了窗口的一条边在另一窗口下十个像素点。一个布置约束的组合可以完全地定义一个窗口的位置(也有一些除外，在下面有所讨论)类 TLayoutMetrics 用了四个布置类约束结构——两个名为 X 和 Y 的 TEdgeconstraint 对象，一个名叫 Width 的 TEdgeOrWidthconstraint 对象，和一个名叫 Height 的 TEdgeOrHeightconstraint 对象。

TLayoutMetrics 是一个十分简单的类，它表示窗口的哪一条边用于约束。X 设置为 ImLeft，Y 设置为 ImTop，Width 设置为 ImWidth，Height 设置为 ImHeight。

一旦构造了 TLayoutMetrics 对象，必须为想要输出的窗口设置布置约束，可以用下面部分叙述的函数设置各项布置约束。

有一点很重要，就是认识到标号 X、Y、Width 和 Height 的使用，比应当严格使用的约束规则更具方便性。X 可以表示窗口左边、右边、或中心点的坐标，可以用约束 Width 组合这些

值,——它可以是 ImCenter、ImRight 或 ImWidth 中的一个——从而完全地定义窗口的 X 轴位置和宽度。用这些边约束比较简单,而且实现时很有用。

最简单的办法是将 X 坐标值赋给 X,将宽度值赋给 Width,但也可能将边的 X 坐标赋给 InCenter,而将边的 Width 赋给 ImRight,所以 Width 不一定真正代表一个宽度,而是窗口右边的 X 坐标,如果知道窗口右边和中间点的 X 坐标,很容易算出窗口左边的 X 坐标。

为了更好地理解约束与描述窗口是如何一起工作的,可以编译并运行目录 EXAMPLES\OWL\OWLAPT\LAYOUT 下的应用程序实例 LAYOUT,这个应用实例中有许多子窗口在布置类窗口中。选择菜单项可以得到一个对话框,改变每个窗口的约束,然后观察窗口输出的结果。注意,虽然定义了许多布置约束,但如果不能完全约束一个窗口,应用程序可能会崩溃,其理由在下面将有讨论。

(三)不确定约束

一定要注意是如何定义布置类窗口的约束的,在类 TLayoutMetrics 中的约束能够完全地描述一个窗口,但是,不能保证实际使用的约束完全地描述了一个窗口,如果出现约束不能完全地描述一个窗口,很可能的结果是程序崩溃。

二、使用布置类窗口

一旦建立了布置类约束,就已经生成了一个布置类窗口,可在其中输出子窗口,下面是类 TLayoutWindow 的构造函数:

```
TLayoutWindow(TWindow * Parent,
              const char far * Title=0,
              TModule * Module=0);
```

其中:

- Parent 是布置类窗口的父窗口。
- Title 是布置类窗口的标题,这个参数默认值是一空字符串。
- Module 作为 TModule 参数传给 TWindow 基类的构造函数,这个参数默认值是 0。

在布置类窗口构造并显示后,有一些函数可以调用:

- 函数 Layout 不带任何参数,并返回 void 类型的值,这个函数告诉布置类窗口寻找其子类窗口并重新显示它们,可以通过调用这个函数,使窗口重新计算其子窗口的边界和位置。通常在移动一个子窗口后,需要调用函数 Layout,重新设置布置类窗口的尺寸,或者其他影响子窗口约束的任何属性。

注意:*TLayoutWindow* 重载了类 *TWindow* 的 *EvSize*,当获得 *WM_SIZE* 事件时自动调用函数 *Layout*。如果自己希望重载这个函数,那么必须保证要么调用这个函数的基类版本,要么调用派生类的函数 *Layout*。

- 函数 SetChildLayoutMetrics 用 TWindow & 和 TLayoutMetrics& 作为其参数,并返

回一个 void 类型值。利用这个函数可以使 TLayoutMetrics 对象中包含的一些约束与子窗口联系起来。下面的例子创建了一个 TLayoutMetrics 对象，并将其与一个子窗口相联系起来：

```
TMyLayoutWindow::TMyLayoutWindow(TWindow * parent,char * title)
:TLayoutWindow(parent,title)
{ TWindow MyChildWindow(this);
  TLayoutMetrics layoutMetrics;
  layoutMetrics. X. Absolute(lmLeft,10)
  layoutMetrics. Y. Absolute(lmTop,10);
  layoutMetrics. Width. PercentOf(lmParent,lmWidth,60);
  layoutMetrics. Height. PercentOf(lmParent,lmHeight,60);
  SetChildLayoutMetrics(MyChildWindow,layoutMetrics);
}
```

在该构造函数中，去创建一个子窗口，然后创建一个尺寸约束对象，该尺寸约束对象用来规定子窗口的位置和大小，调用函数 SetChildLayoutMetrics 将子窗口和约束对象联系起来。

注意：子窗口不需要任何特殊的功能与布置尺寸对象相联系，其联系完全靠布置类窗口本身处理，子窗口不必关心它们的任何关系。

■ GetchildLayoutMetrics 函数带有参数 TWindow& 和 TLayoutMetrics& 并返回 BOOL 类型值，它查看由 TWindow& 指定的子窗口，然后将当前同子窗口关联的布置尺寸放入传递过来的 TLayoutMetrics 对象中。如果 GetChildLayoutMetrics 找不到等于窗口类对象传过来的子窗口，则返回 FALSE。

■ 函数 RemoveChildLayoutMetrics 带一个 TWindow& 参数，并返回 BOOL。它查看由 TWindow& 代表的子窗口，然后从布置窗口的子窗口链表中移走该子窗口及其关联的布置尺寸。如果 RemoveChildLayoutMetrics 不能找到一个子窗口和传递来的窗口对象相等，则返回 FALSE。

必须为布置类窗口中的所有子窗口提供布置类约束，布置类窗口假定其所有子窗口都与布置类分制对象相联系。从布置类窗口中移去子窗口或删去子窗口对象时，会自动地移去相联系的布置类分制对象。

2.2.3 框架窗口

框架窗口(类 TFrameWindow 的对象)是用于支持客户窗口的专门窗口，框架窗口是 MDI 和 SDI 框架窗口，MDI 子窗口连同 TLayoutWindow 和修饰框架窗口的基类窗口。

框架窗口是 ObjectWindows 发展中的一个支柱：框架窗口管理像菜单、工具条等应用范围的任务，在框架内的客户窗口可以定义为专门执行某一任务，在框架窗口中的变化(例如增加工具条和状态条等)不影响客户窗口。

一、构造框架窗口对象

可以用类 TFrameWindow 的两个构造函数中的一个函数构造一个框架窗口对象，这两个构造函数创建新的框架窗口对象连同新的界面元素，并且可将新生成的框架窗口对象与已存在的界面元素相联系起来。

(一)构造一个新的框架窗口

类 TFrameWindow 的第一个构造函数用于创建一个全新的框架窗口对象：

```
TFrameWindow(TWindow * parent,
             const char far * title=0,
             TWindow * clientWind=0,
             BOOL shrinkToclient=FALSE,
             TModule * Module=0);
```

其中：

- 第一个参数是窗口的父类窗口对象，如果正在创建的窗口是主窗口则为 0，(它没有父类窗口对象)，否则用一个指针指向父类窗口对象，这是必须提供的唯一参数。
- 第二个参数是窗口的标题，这是一个显示在窗口的主题条中的字符串，如果不给第二个参数指定任何值，则在标题条中不显示任何标题。
- 第三个参数是为框架窗口定义一个客户窗口。如果第三个参数不指定任何值，则默认情况下，构造函数获得零值，表示无任何客户窗口。否则，传过去一个指向客户窗口对象的指针。
- 第四个参数指明，是否缩小框架窗口去填充客户窗口。如果不定义任何值，默认状态下，构造函数获得值 FALSE，这就意味着不用框架窗填充客户窗口。
- 第五个参数作为 TModule 参数传给基本类构造函数，这个参数的默认值是 0。

这里是使用构造函数的一些例子：

```
void
TMyApplication::InitMainWindow()
{ // default is for no client window
    SetMainWindow(new TFrameWindow(0,"Main Window"));
    }
```

在 TFrameWindow 中，缺省时没有指定客户窗口：

```
void
TMyApplication::InitMainWindow()
{ // client window is TMyClientWindow
    SetMainWindow(new TFrameWindow(0,"Main Window with client";
                  new TMyClientWindow,TRUE));
}
```

在 TFrameWindow 中，客户窗口为 TMyClient；

(二)构造一个框架窗口别名

TFrameWindow 的第二个构造函数用于连接已存在的界面元素和新的 TFrameWindow 对象，这个对象被当作已存在窗口的别名：

```
TFrameWindow(HWND hwnd,TModule * module);
```

其中：

- 第一个参数是已存在界面元素的窗口句柄，这是 TFrameWindow 对象控制的窗口。
- 第二个参数作为构造函数的 TModule 参数传给基类的构造函数，这个参数的默认值为 0。

下面的例子显示了如何为已存在的界面元素构造一个 TFrameWindow 窗口，并将这个窗口作为主窗口：

```
void
TMyApplication:AddWindow(HWND hWind)
{ TFrameWindow frame(hWnd);
  TFrameWindow * tmp=SetMainWindow(frame);
  ShowWindow(GetMainWindow()->HWindow,SW_SHOW);
  tmp->ShutDownWindow();
}
```

构造 TFrameWindow 窗口时，如果使用第二个构造函数，设置标志为 WfAlias。可以通过参数 WfAlias 调用函数 IsFlagSet 来辨别一个窗口元素是否是从窗口对象构造的或者是否是个别名。例如，假设不知道在上面最后例子中的函数 AddWindow 是否执行了，若主窗口不是一个别名，则函数 AddWindow 没有被执行；如果主窗口是个别名，则函数 AddWindow 已经执行。

```
void TMyApplication:: CheckAddExecute()
{ if (GetMainWindow()->IsFlagSet(wfAlias))
  // MainWindow is an alias;AddWindow has executed
  else
  // MainWindow is not an alias;AddWindow
  // has not executed
}
```

二、修改框架窗口

在框架窗口的对象构造后，许多框架窗口的属性可以设置，可以用某些函数改变和询问对象的属性，函数 TFrameWindow 提供了一些附加的设置函数修改框架窗口：

- 函数 AssignMenu 用于在界面元素生成后，设置窗口的菜单，例如在函数 InitMainWindow 中、在窗口对象的构造函数中或函数 SetupWindow 中。
- 函数 SetMenu 将窗口菜单句柄传给参数 HMENU。

■ 函数 SetMenuDescr 将窗口的菜单描述设置给参数 TMenuDescr。
■ 函数 GetMenuDescr 返回当前菜单的描述。
■ 函数 MergeMenu 用参数 TMenuDescr 合并当前的菜单描述。
■ 函数 RestoreMenu 从参数 AttrMenu 中恢复窗口菜单。
■ 函数 SetIcon 设置图标。

程序员可以灵活应用这些函数,创建适合于自己程序需要的框架窗口。TFrameWindow 和另外一些类是 BC++4.0 新增加的类。

2.2.4 修饰过的框架窗口

这一部分讨论修饰过的框架窗口。修饰过的框架窗口包装在类 TDecoratedFrame 中,它是类 TFrameWindow 和 TLayoutWindow 的派生类,修饰过的框架窗口提供所有框架窗口和布置类窗口的功能,另外还提供:

■ 对框架窗口增加控制(称作 decoration)的支持。
■ 自动调整子窗口,提供容纳修饰件的位置。

一、构造修饰过的框架窗口对象

类 TDecoratedFrame 只有一个构造函数,除了第四个参数外,这个构造函数看起来很像前面描述的第一个 TFrameWindow 构造函数。

```
TDecoratedFrame(TWindow * parent,
                const char far * title;
                BOOL trackMenuSelection=FALSE,
                TModule * Module=0);
```

其中:

■ 第一个参数是窗口的父类窗口对象。如果正在创建的窗口是主窗口,则其值为 0(它没有父亲窗口对象),否则,使用一个指向父亲窗口对象的指针,这是需要提供的唯一参数。
■ 第二个参数是窗口的标题,这个字符串显示在窗口的主题条中,如果不给第二个参数指定任何内容,则在标题条中不显示任何标题。
■ 第三个参数要求为框架窗口定义一个指向客户窗口的指针,如果不给第三个参数指定任何值,默认状况下,构造函数获得一个 0 值,这意味着没有客户窗口。
■ 第四个参数指明是否需要追踪菜单项。如果需要追踪,则窗口传给窗口的状态条一个字符串,被传的字符串有一个与被选中的菜单资源相同的名字。除非在窗口中有一状态条,否则不必打开菜单选择项追踪。如果不定义任何值,默认的构造函数获得 FALSE 值,这就意味着不能追踪菜单命令。
■ 第五个参数作为构造函数的 TModule 参数传给基本类的构造函数。这个参数默认值是 0。

二、在修饰的框架窗口中增加修饰项

可以用以前描述的一些方法来修改一个修饰过的框架窗口的属性,类 TDecorated-

Frame 通过使用 Insert 成员函数来提供又一种增加修饰项的能力。

为了使用 Insert 成员函数，首先必须构造一个控制插进来，有效的控制包括控制条(TControlBar)、状态条(TStatusBar)、按钮设备(TButtonGadget)和其他基于 TWindow 的控制类型。

一旦构造了控制，就使用 Insert 函数将这种控制插入修饰框架窗口，Insert 函数需要两个参数：一个对控制的引用和一个定位说明。TDecoratedFrame 提供了 enum 类型的变量 TLocation，TLocation 有四个可能的值：Top，Botton，Left 和 Right。

假设要在修饰过的框架窗口底部增加一个状态条，代码应该是这样的：

```
TStatusBar * sb=new TStatusBar(0,TGadget::Recessed,
                               TStatusBar::CapsLock|
                               TStatusBar:: NumLock|
                               TStatusBar::Overtype);

TDecoratedFrame  * frame=new TDecoratedFrame(0,
                              "Decorated Frame",
                              0,TRUE);
frame-> Insert ( * sb,TDecorated Frame::Bottom);
```

去创建一个状态条，NumLock，CapsLock 等状态将显示在状态条里，再创建一个修饰的框架窗口，将状态条放在框架窗口的底部，BC4.0 集成环境风格与此类似。

2.2.5　MDI 窗口

多个界面或 MDI 窗口是指 MDI 界面的一部分，它用于管理一个应用程序的多个窗口和视口，一个文档通常是一个文件处理任务，如编辑一个文本文件或在一个打开的文件上做工作。

一、*MDI* 应用程序

特定的构件在任意的 MDI 应用程序中都能出现，最好的证据是主窗口，即 MDI frame Window，在框架窗口的客户区域里是 MDI ClientWindow，它包含了称作 MDI 子窗口(MDI Child Windows)的窗口，当使用 Doc/View 类时，应用程序能将视口移入 MDI 窗口。参阅 Doc/View 部分整句改为：可得到关于 Doc/view 类的更详细的信息。

(一)MDI 窗口菜单

一个 MDI 应用程序通过有一菜单项来控制 MDI 子窗口。Window 菜单通常有像 Tile，Cascade，Arrange 和 ClosAll 这样的项，每一个打开 MDI 子窗口的名字将被自动加在菜单的后面，同时检查当前选中的窗口。

(二)MDI 子窗口

MDI 窗口有重迭式窗口的一些特征，一个 MDI 子窗口可以放在 MDI 客户区的整个区域，或缩小至放于客户区域里的一个图标，MDI 子窗口从来不会出现在客户区域框架窗口的外面，尽管 MDI 子窗口没有菜单，但是当一个子窗口被激活时，它们有一个 TMenuDescr

主窗口,可将它用作菜单,每个 MDI 子窗口的标题通常是与窗口相关联的文件的标题。这种方法是可选择的,并且可以控制。

(三)MDI 在 ObjectWindows 里

ObjectWindows 为每种类型的 MDI 窗口都定义了类:

- TMDIFrame
- TMDIClient
- TMDIChild

在 ObjectWindows 中,MDI 框架窗口拥有客户窗口,并且 MDI 客户窗口拥有每一个 MDI 的子窗口。

TMDIFrame 的成员函数管理着框架窗口及其菜单,ObjectWindows 首先将命令传到活动窗口,然后传给其父窗口,所以说客户窗口能处理主窗口的菜单命令,因为 TMDIFrame 没有更多特定的操作,所以很少去派生自己的 MDI 框架窗口类,取而代之的是,仅仅使用一个 TMDIFrame 的实例。即然 TMDIChild 是由 TFrameWindow 派生出来的,它可能是一个带客户窗口的框架窗口,因此可以生成一个特定的窗口作为 TMDIChild 类的特定客户窗口。至于作何种选择,完全依赖于具体的应用程序。

二、建立 *MDI* 应用程序

按照下面步骤可建立一个 ObjectWindows 的 MDI 应用程序:

1. 创建一个 MDI 主窗口。
2. 在一个 MDI 客户窗口上增加行为。
3. 创建 MDI 子窗口。

ObjectWindows 的 TMDIXxx 类处理 MDI 特定的行为,所以设计者可以集中精力对付应用程序指定的行为。

(一)生成一个 MDI 框架窗口

MDI 主窗口通常是一个应用程序的主窗口,所以要在应用程序对象的 InitMainWindow 成员函数里构造它,MDI 框架窗口与其他的框架窗有以下几方面的不同:

- 一个 MDI 框架主窗口总是主窗口,它没有父窗口,因此,TMDIFrame 的构造函数不使用指向其父类窗口对象的指针作为参数。
- 一个 MDI 框架必须有一个菜单,所以 TMDIFrame 的构造函数使用一个菜单资源标识符作为参数,对于非 MDI 主窗口而言,必须调用 AssignMenu 来设置窗口菜单。而 TMDIFrame 的构造函数已经完成这些工作。TMDIFrame::AssignMenu 所做工作的一部分就是为子窗口菜单搜索这个菜单,搜索方法是通过寻找特定的菜单命令 IDs 来完成的,如果发现了一个 Window 菜单,新的子窗口标题将自动加到菜单的最下方。

一个典型的 MDI 应用程序的 InitMainWindow 函数定义如下:

```
void:
TMDIApp::InitMainWindow()
{
    SetMainWindow(new TMDIFrame("MDI App",ID_MENU,
                    * new TMyMDIClient));
}
```

这个例子产生了一个从 ID_MENU 资源获得的菜单并具有标题"MDIApp"的 MDI 框架窗口。菜单 ID_MENU 应有一个子窗口菜单,MDI 的客户窗口是由类 TMyMDIClient 来创造的。

(二)在 **MDI** 客户窗口里增加操作

由于经常使 TMDIFrame 的一个实例作为 MDI 框架窗口,所以必须在 MDI 客户窗口中加入应用程序方面的操作,框架窗口拥有菜单和工具条,但是它将产生的命令送至客户窗口和应用程序中,一个公共的消息响应函数将响应 File|Open 菜单以打开另一个 MDI 子窗口。

操作子窗口

TMDIClient 有几个成员函数来操作 MDI 子窗口,MDI 应用程序的子窗口菜单命令控制子窗口。TMDIClient 自动地对那些命令做出响应,并执行相应的动作。

表2.3 标准的 MDI 子窗口的菜单行为

动 作	菜单命令 ID	TMDIClient 成员函数
Cascade	CM_CASCADECHILDREN	CmCascadeChildren
Tile	CM_TILECHILDREN	CmTileChildren
TileHorizontally	CM_TILECHILDRENHORIZ	CmTileChildrenHoriz
Arrange Icons	CM_ARRANGEICONS	CmArrangeIcons
Close All	CM_CLOSECHILDREN	CmCloseChildren

头文件 OW\mdi. h 包含了应用程序的头文件 OWL\mdi. rh。OWL\mdi. rh 是一个资源头文件,它定义了以上所列的菜单命令 IDs。在资源底稿中设计菜单时,要确认包含了 OWL\mdi. rh。以获得这 IDs 值。

MDI 子窗口不应该对任意子窗口菜单命令都做出响应,而由 MDI 客户窗口关心它们。

(三)生成 **MDI** 子窗口

生成 MDI 子窗口有两种方法:在 TMDIClient::Initchild 中自动生成和在任何地方手工生成。

自动生成子窗口

TMDIClient 定义了 CmCreateChild 消息响应函数来响应消息 CM_CREATECHILD,CmCreateChild 通常是用来响应一个 MDI 应用程序的 File|New 菜单命令。CmCreateChild 调用 CreateChild,CreateChild 调用 InitChild 来构造一个 MDI 子窗口对象,最后调用那个对象的成员函数 Create 来产生一个 MDI 子窗口的界面元素。

如果 MDI 应用程序使用 CM_CREATECHILD 作为命令 ID 来创建新的 MDI 子窗口，那么必须在 MDI 客户窗口类中重载函数 InitChild，这样就可以在用户选择了那条命令时，构造 MDI 子窗口类对象：

```
TMDIChild *
TMyMDIClient::InitChild()
{
    return new TMDIChild(*this,"MDI child Window");
}
```

由于 TMDIChild 的构造函数用一个指向其父窗口对象的引用作为参数，而不是一个指针。

手工生成子窗口类

不一定非得在函数 InitChild 中构造 MDI 子窗口对象。不过，如果在别的地方构造，就必须自己生成它们的界面元素：

```
void
    TMyMDIClient::CmFileOpen()
        {
            new TMDIChild(*this, * *)->Create();
        }
```

2.3 事件处理

这一节介绍如何使用 ObjectWindows 2.0的响应表。响应表是 ObjectWindows 2.0应用程序中处理所有事件的途径。使用 ObjectWindows 2.0的响应表有四个主要的步骤：

1. 声明响应表；
2. 定义响应表；
3. 定义响应表的各项；
4. 声明和定义响应表的成员函数。

使用本章描述的所有宏，都必须包含头文件 OWL\eventhan.h，这个文件已经包含在头文件 OWL\module.h(它包含在头文件 OWL\applicat.h 中)和头文件 OWL\window.h 中，所以通常没有必要明确地包含这个头文件。

ObjectWindows 2.0的响应表，比起其他的处理 Window 事件和消息的方法来说，是一种重要的改进，在一般的应用程序中，通常使用 switch 语句和方案(像标准 C Windows 程序中)，与其他的处理事件方法不同，ObjectWindows 2.0响应表提供了如下特点：

- 预定义命令消息的自动分解功能，因此减少了手工分解参数 WPARAM 和 LPARAM 值的需要。
- 编译时错误和类型检查，检查事件处理函数的返回类型和参数类型。
- 一个函数处理多条消息的能力。

- 支持多重继承，每个派生类都建立在顶层基本类或类的响应表上。
- 由于不依赖特定的编译扩展名，从而具有良好的平台可移植性。
- 使用预定义的响应表宏，很容易处理命令、子类 ID 通知和客户消息。

2.3.1 声明响应表

由于响应表是一个 ObjectWindows 类的一个成员，所以在定义类时，必须声明响应表。ObjectWindows 提供宏 DECLARE_RESPONSE-TABLE 隐藏了响应表使用的真正的语法模型。

宏 DECLARE_RESPONSE_TABLE 只有一个参数，响应表声明时使用对应的类的名字，将这个宏放在类定义的最后。例如，类 TMyFrame 是类 TFrameWindow 的派生类，是这样定义的：

```
class TMyFrame:public TFrameWindow
{
    ⋮
    DECLARE_RESPONSE_TABLE(TMyFrame);
};
```

2.3.2 定义响应表

一旦声明了响应表，就必须定义它，响应表必须在类定义的外面定义。

ObjectWindows 是用宏 DEFINE_RESPONSE_TABLEX 来定义响应表的，X 的值依赖于类的继承性，其值等于这个类具有的直接基本类的值，宏 END_RESPONSE_TABLE 结束事件响应表的定义。

为了定义响应表，应做到以下几步：

1. 用宏 DEFINE_RESPONSE_TABLEX 开始类的响应表的定义，宏 DEFINE_RESPONSE_TABLEX 有 x+1个参数：
   - 当前定义的响应表所对应的类名。
   - 每个直接基本类的名。
2. 填充响应表的各项(具体实现这一步参见下一节)。
3. 用宏 END_RESPONSE_TABLE 结束响应表的定义，例如，类 TMyFrame 的响应表的定义，它是类 TFrameWindow 的派生类，应该这样定义：

```
DEFINE_RESTORE_TABLE(TMyFrame,TFrameWindow)
    EV_WM_LBUTTONDOWN,
    EV_WM_LBUTTONUP,
    EV_WM_MOUSEMOVE,
    EV_WM_RBUTTONDOWN,
END_RESTORE_TABLE;
```

必须在每个响应表项的后面放置一个逗号，并在宏 END_RESPONSE_TABLE 后面放置一个分号。

2.3.3 定义响应表的项

响应表的各项是通过特定的函数与 Windows 事件相联系的，当一个窗口或控制接受一条消息，它就检查其响应表，看是否有与这个消息相对应的项，如果有，就将这条消息传给相应的函数，如果没有，就将消息送给其父类。如果这个窗口是主窗口，则将消息传给应用程序对象，如果应用程序对象没有与那条消息相应的响应项，则这条消息送到 ObjectWindows 的默认过程处理。

ObjectWindows 为响应表的提供了很多数目的宏。它们包括：

- 命令消息宏，可以处理命令消息，并将他们传给特定的函数。
- 标准的 Windows 消息宏，处理 Windows 消息。
- 已登记的消息(函数 RegisterWindowMessage 返回的消息)。
- 子类 ID 通知宏，在子表或父类中处理子类 ID 通知码。
- 控制通知宏，可以处理来自专门的控制的消息，像按钮、组合框、编辑控制、列表框等等。
- 文本管理器宏通知应用程序，已经创建了一个文本或视口，或者已经消除，并且通知视口关于文本管理器方式的消息。
- VBX 控制通知。

一、命令消息宏

ObjectWindows 提供了大量名叫命令消息宏(command message macros)的宏，可让设计者指定命令消息的任何函数。唯一的要求是在处理某一消息的函数中的定义必须同对应消息的宏所要求的保持一致。不同命令消息的宏列于下表中：

表2.4 命令消息宏

宏	原 型	描 述
EV_COMMAND (CMD,UserName)	void UserName()	收到命令消息 CMD 后，调用成员函数 UserName
EV_COMMAND_AND_ID (CMD,UserName)	void UserName (WPARAM)	收到命令消息 CMD 后，调用成员函数 UserName，将命令 ID(即参数 WPARAM)传给这个函数
EV_COMMAND_ENABLE (CMD,UserName)	void UserName (TCommand Enable&)	用于自动的使命令控制像菜单项，工具条按钮等有效或失效

还有其他的消息宏可以向事件处理函数传递原始的、未处理的消息，这些消息宏处理任何类型的通用消息和记事消息。

表2.5 消息宏

宏	原 型	描 述
EV_MESSAGE (MSG,UserName)	LRESUT UserName (WPARAM,LPARAM)	当接受到用户定义的消息 MSG 时调用成员函数 UserName,MSG 未经修改传给函数 UserName
EV_REGISTEPED (MSG,UserName)	LRESUT UserName (WPARAM,LPARAM)	接受登记消息时,调用成员函数 UserName,消息 MSG 未经修改传给函数 UserName

正确地用在响应表中定义的宏去匹配函数的定义是很重要的。例如,假定有下面的代码:

```
Class TMyFrame:Public TFrameWindow{
public;
    void CmdAdvise();
    DECLARE_RESPONSE_TABLE(TMyFrame);
};
DEFINE_RESPONSE_TABLE(TMyFrame,TFrameWindow)
        EV_COMMAND_AND_ID(CM_ADVISE,CmAdvise),
END_RESPONSE_TABLE;
void TMyFrame:: CmdAdvise() {
  ⋮
}
```

这段代码在编译时会出错,因为宏 EV_COMMAND_AND_ID 要求一个返回 void 类型的函数,并且只有一个参数 WPARAM。在这个例子里,函数正确地返回 void 类型,而不带任何参数是不正确的。为了使这段代码能正确编译,改变成员声明和函数 TMyFrame::CmAdvise 的定义如下:

```
void TMyFrame::CmAdvise(WPARAM cmd);
```

二、*Windows* 的消息宏

ObjectWindows 为所有的标准的 Windows 消息提供预定义宏,可以在一个类成员函数中用这些宏去处理标准的 Windows 消息。为了寻找宏的名字,在消息名前加上 EV,例如,处理消息 WM_PAINT 的宏是 EV_WM_PAINT,处理消息 WM_LBUTTONDOWN 的宏是 EV_WM_LBUTTONDOWN。

这些预定义的宏将消息用预定义的名字传给函数,决定函数名时,从消息名上除去 WM_将 EV 加到消息名剩下部分,并将每个单词除首字母外大小变小写。例如,消息 WM_LBUTTONDOWN 传给名叫 EvLButtonDown 的函数。

用 Windows 消息宏的优点是,Windows 消息宏能自动"分解",也就是说,通常经过译码存储在变量 PARAM 和 WPARAM 中的参数,放在固定的位置,并作为独立的参数传给事件处理函数。

例如,宏 EV_WM_CTLCOLOR 将分解后的参数传给用下面的名字定义的事件处理函数:

```
HBRUSH EvCtlColor(HDC hDCChild,HWND hWndChild,UINT nCtrlType);
```

消息分解提供严格地C++编译时类型检查,并帮助设计者在编译代码时而不是在运行时发现错误。它还有助于应用程序从十六位过渡到三十二位应用,反之亦然。在 ObjectWindows Reference Guide 一书的第二章里列出了每个窗口消息,它对应于响应表中的宏,并对应于事件处理函数的取名。

为了使用预定义的 Windows 消息宏,在响应表中增加宏,并且在类中增加相应取名的成员函数,例如,假如类对象 TMyFrame 接受消息 WM_ERASEBKGND 时执行一些操作,其代码应该这样定义:

```
Class TMyFrame:public TFrameWindow
{ public:
  BOOL EvEraseBkgnd(HDC);
  DECLARE_RESPONSE_TABLE(TMyFrame);
};
DEFINE_RESPONSE_TABLE(TMyFrame,TFrameWindow)
    EV_WM_ERASEBKGND,
END_RESPONSE_TABLE;
BOOL TMyFrame::EvEraseBkgnd(HDC hdc)
{ ...
}
```

从 TFrameWindow 类以公有方式派生出 TMyFrame 新类,为了在该类中处理预定义消息 WM_ERASEBKGND,必须为 TMyFrame 声明一个事件响应表,这在类定义中用 DECLARE_RESPONSE_TABLE 变成,在类定义之外,定义 EV_WM_ERASEBKGND,Borlandc c++ OWL 自动将该消息分发给 EvEraseBkgnd 函数处理。

三、子类 *ID* 通知消息宏

子类 ID 通知消息宏提供了许多不同的方法来处理子类 ID 通知消息。可以从带有单个函数的多重子类中处理通知码,将所有来自于子类的通知码传给一个响应窗口,或者在子类中处理通知码。

可以利用这些宏使得对子类的控制和通信更方便,不同的子类 ID 通知消息宏列于下表中。

表2.6 子类 ID 消息通知宏

宏	原 型	描 述
EV_CHILD_NOTIFY (ID,Code,UserName)	voidUserName()	将消息和通知码分配给成员函数 UserName
EV_CHILD_NOTIFY_AND_CODE (Id,Code,UererName)	void UserName (WPARAM code)	将消息 ID 用通知码 Code 传给函数 UserName
EV_CHILD_NOTIFY_ALL_CODES (Id,UserName)	void UserName (WPARAM code)	将消息 Id 传给函数 UserName 而不管消息通知码
EV_CHILD_NOTIFY_AT_CHILD (Code,UserName)	void UserName()	将通知码 Code 分配给子类对象的成员函数 UserName

这些宏提供了不同的方法来处理子类 ID 通知码。如果希望在子类的父窗口中处理子类 ID 通知码，用宏 EV_CHILD_NOTIFY，它将通知码作为一个参数传递，并用一个单一的函数处理多重的子类 ID 通知码，这也防止对于每一个控制，在分散的响应表中处理每个子类的通知消息；相反，每条消息在父类中被处理，例如，在响应表中使一个对话框处理所有的控制。

例如，假如你有一个名叫 TTestDialog 的对话框，有四个按钮，它们的 IDs 是 ID_BUTTON1，ID_BUTTON2，ID_BUTTON3，和 ID_BUTTON4。当用户点中其中一个按钮时，希望有一个单独的函数来处理这条消息，而不管是哪个按钮被按下。如果用户双击按钮，希望用另一个专门的函数来处理这个事件。这段代应该这样。

```
class TTestDialog:public TDialog
{ public:
    TTEstDialog(TWindow * parent,TResId resId);
    void HandleClick();
    void HandleDbclick1();
    void HandleDbClick2();
    void HandleDbclick3();
    void HandleDbClick4();
  DECLARE_RESPONSE_TABLE(TTestDialog);
};
DEFINE_RESPONSE_TABLE1(TTestDialog,TDialog)
    EV_CHILD_NOTIFY(ID_BUTTON1,BN_CLICKED,HandleClick),
    EV_CHILD_NOTIFY(ID_BUTTON2,BN_CLICKED,HandleClick),
    EV_CHILD_NOTIFY(ID_BUTTON3,BN_CLICKED,HandleClick),
    EV_CHILD_NOTIFY(ID_BUTTON4,BN_CLICKED,HandleClick),
    EV_CHILD_NOTIFY(ID_BUTTON1,BN_DOUBLECLICKED,HandleClick1),
    EV_CHILD_NOTIFY(ID_BUTTON2,BN_DOUBLECLICKED,HandleClick2),
    EV_CHILD_NOTIFY(ID_BUTTON3,BN_DOUBLECLICKED,HandleClick3),
    EV_CHILD_NOTIFY(ID_BUTTON4,BN_DOUBLECLICKED,HandleClick4),
END_RESPONSE_TABLE;
```

```
DEFINE_RESPONSE_TABLE1(TTestWindow,TWindow)
    EV_COMMAND(ID_BUTTON,HandleButtonMsg),
    EV_COMMAND(ID_CHECKBOX,HandleCheckBoxMsg),
    EV_CHILD_NOTIFY_ALL_CODES(ID_GROUPBOX,HandleGroupMsg),
END_RESPONSE_TABLE;
```

如果希望所有来自于子类的通知码都传给父窗口,用宏 EV_CHILD_NOTIF_ALL_CODES,它是子类 ID 通知码的句柄。例如范例程序 BUTTONX.CPP 中定义了如下的响应表:

```
DEFINE_RESPONSE_TABLE1CTTestWindow,TWindow))
    EV_COMMAND(ID_BUTTON,HandleButtonMsg),
    EV_COMMAND(ID_CHECKBOX,HandleCheckBoxMsg),
    EV_CHILD_NOTIFY_ALL_CODES(ID_GROUPBOX,HandleGroupBoxMsg),
END_RESPONSE_TABLE
```

这个表处理按钮、复选框、成组框的消息。这种情况下,父类窗口(TTextWindow)获得所有从子类(ID_GROUPBOX)传来的通知消息。

宏 EV_CHILD_NOTIFY_ALL_CODE 用用户定义的函数 HandleGroupBoxMsg 处理这些消息。结果,如果用户在成组框中击中按钮,一个消息框便会出现并告诉用户哪一个按钮击中。

如果希望父窗口用同一函数处理多于一条的消息,可以使用宏 EV_CHILD_NOTIFY_AND_CODE。

例如:

```
DEFINE_RESPONSE_TABLE1(TTestWindow,TWindow).
EV_CHILD_NOTIFY_AND_CODE(ID_GROUPBOX,SomeNotifyCode,
                         HandleThisMessage),
EV_CHILD_NOTIFY_AND_CODE(ID_GROUPBOX,AnotherNotifyCode,
                         HandleThisMessage),
END_RESPONSE_TABLE;
```

如果窗口中有多条不同的消息需要处理,并且是用不同的函数处理这些消息,最好使用宏 EV_CHILD_NOTIFY_AND_CODE,而不用 EV_CHILD_NOTIFY,因为宏 EV_CHILD_NOTIFY 的消息处理函数不接受参数,所以不知道哪一条消息正在处理。

要在子类窗口中处理子类 ID 通知消息,用宏 EV_CHILD_NOTIFY_AT_CHILD。范例程序 NOTITEST,CPP,中包含了下面的响应表:

```
DEFINE_RESPONSE_TABLE1(TBeepButton,TButton)
    EV_NOTIFY_AT_CHILD(BN_CLICKED,BnClicked),
END_RESPONSE_TABLE;
```

这个响应表用宏 EV_NOTIFY_AT_CHILD 通知子类窗口(TBeepBuuton)用函数 BnClicked 去处理通知码(BN_CLICKED)。

2.4 对话框对象

对话框对象是一种封装了对话框功能的界面对象。TDialog 类直持各种类型对话框的初始化、创建和执行。同从 TWindow 类派生各种窗口一样，在应用程序当中，可以从 TDialog 类派生各种特殊类型的对话框对象。

ObjectWindows 也提供了通用对话框窗口类，该类为用户提供了一种选择文件名、字体、颜色等的简单方法。

本节将讨论如下几个主题：

- 应用对话框对象
- 将对话框作为主窗口
- 在对话框中使用控制
- 关联界面对象与控制
- 应用通用对话框

2.4.1 应用对话框对象

应用对话框对象与应用窗口对象的方法非常相似。因为对话框对象常常停留很短的一段时间，可用父窗口对象的一个成员函数控制对话框的创建、执行和撤销。

应用对话框对象的步骤是：

- 创建对话框对象
- 执行对话框对象
- 关闭对话框
- 析构对话框

一、创建对话框对象

设计和创建对话框需要对话框资源，为了建立对话框资源，可以用 Borland 公司提供 Resource Workshop 或其他的资源编辑器。对话框资源描述了各种控制的外观及位置，如按钮、列表框、成组框等。对话框资源并不负责对话框的动作，这一点由应用程序来负责。

每一个对话框资源，都有一个用来联系对话框对象与其所用资源的标识符，该标识符既可以是一个字符串，也可以是一个整型数。它被传给对话框的构造函数，用以指明创建指定的对话框时应当使用的对话框资源。

创建对话框的构造函数带两个参数：一个指向父窗上对象的指针，一个资源标识符。

```
TDialog dialog1(this,"DIALOG-I");
:
TDialog dialog2(this,IDD_MY_DIALOG);
```

由于经常是在一个窗口内创建对话框，因此用 this 作为指向父窗口的指针。如果不是在一个窗口内创建对话框，则此时的父窗为应用程序的主窗口，因为它是 ObjectWindows 应用程序中唯一的一个窗口对象：

```
TDialog mySpecialDialog(GetApplication()->GetMainWindow();IDD_DLG);
```

例外的情况是：在 TFrameWindow 或 TFrameWindow 派生类的构造函数中，把对话框对象指定为客户窗口。该构造函数把对话框对象传递给 TFrameWindow::Init 函数，在 Init 函数中，对话框对象被设置为父窗口。

二、执行对话框

执行一个对话框与创建并显示一个窗口类似，由对话框停留的时间一般都比较短，根据对话框是有模式对话框还是无模式对话框，有些步骤可以得到简化。

(一)有模式对话框

大部分对话框都是有模式的。当显示一个有模式对话框时，它将冻结与该对话框无关的所有操作。用户只能应用该对话框，或者关闭该对话框，而在关闭该对话框之前，用户无法进行其他的工作。

TDialog::Execute 函数执行一个有模式的对话框。当用户关闭执行的对话框时，Execute 函数返回一个反映用户如何关闭对话框的整型数。该整型数是用户按下的控制的标识，例如，OK 按钮的 IDOK，"Cancet"按钮的 IDCANCEL，另外需要注意，如果对话框对象是动态创建的，必须用 delete 将其删除。

在下面的例子中，IDD_MY_DIALOG 为对话框资源的标识符，该对话框有两个按钮按下"OK"按钮，返回值为 IDOK，按下"Cancel"按钮，返回其他的值：

```
if(TMyDialog (this IDD_MY_DIALOG).Execute()==IDOK)
    // user pressed OK
else
    // user pressed cancel
```

只有将对话框对象删除掉，它才真正消失，但资源仍然存在。可以仅利用一个对话框资源创建和删除任意多个对话框。

(二)无模式对话框

无模式对话框与有模式对话框不同，它不冻结其他的操作。在一个无模式的对话框尚未关闭的情况下，用户仍可继续执行其他的动作。

TDialog::Create 执行无模式的对话框。当用 Create 函数创建无模式对话框时，要想使对话框可见，必须明确指明资源具有 WS_VISIBLE 属性，或者调用 ShowWindow 函数。

例如，假设有下面的资源文件：

```
DIALOG_I DIALOG 18,18,142,44
STYLE DS_MODALFRAME |WS_POPUPLWS_CAPTION|WS_SYSMENU
CAPTION"Dialog 1
{
    PUSHBUTTON"Button",IDOK,58,23,25,16
}
```

利用上述资源建立一个无模式对话框,代码如下:

```
⋮
TDialog dialog1(this,"DIALBG_1");
dialog1.Create();
⋮
```

上面创建的对话框不会显示在屏幕上,为了使之显示,必须做下面两件事之一:

■ 改变对话框资源的属性,使之包括 WS_VISIBLE:

```
STYLE DS_MODALFRAME|WS_POPUP|
    WS_CAPTION|WS_SYSMENU|WS_VISIBLE
```

■ 调用 Create 函数之后接着调用 ShowWindow:

```
⋮
TDialog dialog1(this,"DIALOG_1");
dialog1.Create();
dialog1.ShowWindow(SW_SHOW);
⋮
```

TDialog 类的 CmOk 函数和 CmCancel 函数可以关闭及删除对话框对象,他们分别处理来自于"OK"按钮的 IDOK 和来自于"Cancel"按钮的 IDCANCEL 消息。CmOk 函数调用 CloseWindow 函数关闭对话框;CmCancel 函数调用带 IDCANCEL 参数的 Destroy 函数。这两个函数都具有关闭对话框的功能。如果用户欲重载 CmOk 和 Cancel 函数,必须先调用基类的 CmOk 和 CmCancel,然后再执行关闭,清除操作。

另外还有一种创建对话框对象的方法,在父对话框的构造函数中创建一个子对话框。按照这种方法,子对话框对象只需创建一次,而且,对这个对话框对象所做的任何改变在用户下次打开对话框时都会得到保存。

与子窗口一样,在父对话框被撤销的时候,子对话框被自动删除。因此没有必要明确地删除子对话框对象。

在下面的程序代码中,一个父窗口对象的构造函数构造了一个对话框对象,其他的函数来创建、显示这个无模式的对话框:

```
class TParentWindow:public TFrameWindow
{ public:
    TParentWindow (TWindow * parent,const char * title);
    void CmDOIT();
    protected:
    TDialog  * dialog;
};
⋮
void TParentWindow::CmDoIt()
{ dialog=new TDialog (this,IDD_EMPLOYEE_INFO);
  dialog->create();
```

```
}
```

(三)应用对话框的自动创建功能

如果设置了自动生成标置，则 ObjectWindows 会自动创建一系列子对话框对象。在从 Window 派生类的父窗口对象的构造函数中创建子对话框对象，把 this 指针作为指向父窗口的指针，或者是把对话框对象作为父窗口对象的一个成员，在父窗口创建时，它将检查出子对象列表中具有 wfAutoCreate 标志置位的子对象，依次创建这些子对象。这样，在父窗口显示在屏幕上时，子窗口或子对话框将随之显示在屏幕上。

调用 EnableAutoCreate 函数可以置位 wfAutoCreate 标志，调用 DisableAutoCreate 函数清除，wfAutoCreate 标志。

与创建无模式的对话框类似，可以利用自动生成标去来创建一个有模式对话框，但对话框的资源文件必须包含 WS_VISIBLE 属性。ShowWindow 函数不能用来使能自动生成功能。

下面的代码演示了如何用自动生成功能创建一个有模式对话框：

```
class TMyFrame:public TFrameWindow
{ public:
     Tdialog * dialog;
     TMyFrame(TWindow * ,const char far * title);
};
TMyFrame: TMyFrame (TWindow * parent,const char * title).
{ Init (parent,title,TRUE);
     dialog=new TDialog(this"MYDIALOG");
     // For the next line to work property ,the WS_VISIBLE.
     // atribute must be specified for MYDIALOG resource
     dialog-> EnableAutoCreate();
}
```

执行这个应用程序时，对话框自动创建。

(四)管理对话框

对话框与其他的子窗口，如窗口、控制等是有区别的，即对话框在父窗口的生存期中，要经常地被显示和删除，几乎不随主窗口一同显示和删除。通常，只有当用户选择了一个菜单选项，按下了鼠标按钮，出现了异常情况或发生了其他的事件，应用程序才会弹出一个对话框。

在创建新的对话框对象时，必须删除先前的对话框对象。在父窗口的构造函数中构造对话框对象，或者把对话框对象作为父窗口对象的数据成员来创建对话框对象时，必须牢记这一点。这些对话框对象被插入到子窗口列表中，当父窗口被删除时这些子对象被自动删除。

只要对话框对象仍然存在，在任何时候，它都能返回一个值。例如若用户按下了"OK"按钮，返回 IDOK，该返回值在 CmOK 函数中处理。

处理错误的对话框

与窗口对象一样，对话框对象的成员函数 Create 和 Execute 可以设置一个意外捕捉类 TXWindow。当由于指定的资源不存在或者内存空间不够而导致对话框不能创建时，将执行这一个意外捕捉类。

当应用程序开始处理 TXWindow 时，ObjectWindows 为其设置一个全局的意外事件句柄。借助于该句柄可进行适当的错误处理。另外，也可以安装用户自定义意外事件句柄，为了设置用户自定义的句柄，用 try/catch 块包含欲保护的程序代码。例如，如果欲知道 DoStuff 是否会产生错误，可利用下面的代码。

```
try {
    DoStuff();
}
Catch (TWindow::TXWindow & e){
    MessageBox(0,e.why().C_Str(),
        "Error",MB_OK);
}
```

三、关闭对话框

每一个对话框都必须向用户提供一个将其关闭的方法。对于有模式的对话框，通常有"OK"或"Cancel"按钮，或者两者都有。TDialog 类具有相应的事件处理函数 CmOk 和 CmCancel。

CmOK 调用 CloseWindow 函数，CloseWindow 调用 CanClose 确定是否能关闭指定的对话框。如果 CanClose 返回值 TRUE，CloseWindow 传输对话框数据并关闭对话框。

CmCancel 函数调用 Destroy 关闭对话框。不检查对话框是否能关闭，也不进行数据传输。

为了检查对话框的输入，需要重载对话对象的成员函数 CanClose，在重载 CanClose 时，一定要调用父窗口的 TWindow::CanClose 函数，它负责调用所有子窗口的 CanClose 函数。

2.4.2　将对话框作为主窗口

为了将对话框作为主窗口，最好是把主窗设置成框架窗口，对话框作为框架窗口的客户窗口。为了作到这一步，需要从 TApplication 类派生一个应用类，并提供一个成员函数 InitMainWindow。在 InitMainWindow 中，构造框架窗口对象，把对话框指定为客户窗口。

TFrameWindow 的构造函数带有五个参数，第三个参数是一个指针，指向客户窗口。实现的代码如下：

```
#include<owl\applicat.h>
#include<owl\framewin.h>
#include<owl\dialog.h>
class TMyApp:public TApplication
{ public:
```

```
  TMYApp (char * title):TApplication(title){}
  void InitMainWindow();
};
Void TMyApp::InitMainWindow()
{ SetMainWindow(new TFrameWindow(0,"My App",new TDialog(0",MyDialog"),TRUE));
}
int OwcMain(int argc,char * * argv)
{ return TMyApp("MyApp").Run();
}
```

TFrameWindow 的构造函数为作为客户窗口的对话框对象设置自动生成标志。不管传给它的状态是什么。

必须确保对话框资源具有如下属性：

- 析构对话框对象，而不析构框架窗口。框架窗口需要单独明确析构。
- 因为对话框不是主窗口，不能为它动态地加入资源，而必须将资源赋给框架窗口。例如，假设以前用 SetIcon 函数为对话框加入了一个图标，现在必须用 SetIcon 函数为框架窗口设置图标。
- 必须通过框架窗口设置标题，而不能在资源文件中指定。
- 必须设置对话框具有下述属性：
  - 可见(WS_VISIBLE)
  - 子窗口(WS_CHILD)
  - 禁止极小化、极大化、按钮出现，不允许拖动、不允许有系统菜单等其他标准框架窗口所有的属性。

2.4.3 在对话中应用控制

几乎所有的对话框都包含有控制，如编辑控制、列表框、按钮等。这些控制在对话框的资源中定制。

对话框与其包含的控制可进行双向通讯、一方面，对话框能够操纵其包含的控制，例如对话框可向其中的列表框增加列表项。另一方面，对话框能对控制发出的消息提供响应并进行一定的处理。

与控制通讯

Windows 定义了一组消息，应用程序可将这些消息发送给 Windows，实现与 Windows 的通讯。例如列表框消息包括 LB_GETTEXT，LB_GETCURSEL，LB_ADDSTRING。控制消息包含了具体的控制及消息的具体内容，消息的内容包含在 wParam 和 lParam 两个参数中。对话框资源中的每个控制都有一个标识符，它用来指明接受某一消息的控制。调用 SendDlgItemMessage 可发送一个控制消息。例如在下面的示例中，该函数利用 LB_ADDSIRING 消息向列表框中增加一字符串：

```
void TTestDialog::FillListBox(const char far * string)
{
```

```
    SendDlgItemMessage(ID_LISTBOX,LB_ADDSTRING,0,(LPARAM)String);
}
```

尽管 TListBox::AddString 与 SendDtgItemMessage 完成一致的功能。而且较易于理解，但这里是为了告诉读者如何应用 SendDegItemMessage 函数。

很少有必要如上述那样和控制通信，ObjectWindows 控制类提供的成员函数可以完成相同的功能。这一部分只是为了加深理解，才讨论这种实现能信的机理。

2.4.4 关联界面对象与控制

由于对话框的外观及其所包含的控制都是在资源文件中定制的。程序员不能用 C++ 代码，对其进行描述。尽管这种方法可能会显得很直观，但它将使你难于在应用程序中应用这些控制。为此，ObjectWindows 为程序员提供关联对话框中的控制与界面对象的方法，为实现关联，需要做下面两件事：

- 为消息提供特殊的响应。例如，希望一个编辑控制仅接受数字输入；或者希望一个按钮被按下时改变风格。
- 借助于数据成员和成员函数应用控制。这种方法比应用控制消息更简单更面向对象。

一、控制对象

为了关联控制对象与控制元素，需要在对话框对象的构造函数中构造一个控制对象，定义一个指向控制对象的指针。控制类，如 TButton，其构造函数带有两个参数，一个是指向父窗口的指针，一个是资源标识。在下面的例子中，TTestDialog 的构造函数利用资源 ID_BUTTON 创建了一个 TButton 对象：

```
TTestDialog::TTestDialog(TWindow * parent,char * resID)
:TDialog(parent,resID),TWindow(parent)
{ new TButton(this,ID_BUTTON);
}
```

从已存在的控制类，可以定义自己的控制类。在下面的例子中，TBeepButton 类从 TButton 类派生出来的具有特殊功能的类，它重载了对 BN_CLICKED 消息的缺省响应。一个 TBeepButton 对象与 ID_BUTTON 资源相关联：

```
class TBeepButton:public TButton{
    public:
        TBeepButton(TWindow * parent, int resId):TButton(parent,resId){}
            // child id notification handled at the child
            //
            void BNClicked();       // BN_CLICKED
    DECLARE_RESPONSE_TABLE(TBeepButton);
};

DEFINE_RESPONSE_TABLE1(TBeepButton, TButton)
```

```
    EV_NOTIFY_AT_CHILD(BN_CLICKED,BNClicked),
    END_RESPONSE_TABLE;
void
TBeepButton::BNClicked()
{
    MessageBeep(0)
    DefaultProcessing();            // pass it along to parent.
}
TBeepDialog::TBeepDialog(TWindow * parent,TResID resId)
    :TDialog(parent,resId),
     TWindow(parent)
{
    BeepButton=new TBeepButton(this,ID_BUTTON);
}
```

与初始化窗口对象不同,[初始化窗口对象需要两步,(构造和生成)],关联界面对象与界面元素仅需构造一步,这是因为界面元素已经存在,它从资源文件中装入。编程人员只需要应用资源标识符通知构造函数,该函数控制着资源的使用。

二、建立控制

在执行对话框对象的 SetupWindow 成员函数之前,控制还不能被使用,如向列表框中加入字符串,设置编辑控制的字体等。在 TDialog::SetupWindow 调用 TWindow::SetupWindow 之前,对话框的控制还没有与相应的对象关联。只有在建立了关联以后,对象的 HWindow 数据成员才会是一个有效的值。

在下面的例子中,AddString 函数是在调用了 SetupWindow 函数之后被调用的:

```
class TDerivedDialog:public TDialog
{ public:
    TDerivedDialog (TWindow * parent, TResId resId)
    :TDialog(parent,resId),TWindow(parent)
    { listbox=new TListBox(this,IDD_LISTBOX);
    }
    protected:
      TListBox * listbox;
};
void TDerivedDialog::SetupWindow()
{ TDialog::SetupWindow();
    listbox->AddString("first entry");
}
```

从 TDialog 派生出一个派生的对话框,在类定义中,保留一个列表框的指针,调用 SetupWindow 函数来初始化该对话框时,必须先调用基类 TDialog 的 SetupWindow 函数。然后调用列表框的 AddString 向列表框中增加项目。

2.4.5 应用对话框

在 Windows 的应用程序当中，经常要提示用户输入文件、选择颜色、设定字体度。ObjectWindows 为此提供了一种非常简单的方法——应用对话框，包括 Windows 的通用对话框。下面列出了 ObjectWindows 类封装的各种不同类型的对话框：

表2.7 ObjectWindows 封装的对话框

类 型	ObjectWindows 类
Color	TChooseColorDialog
Font	TChooseFontDialog
FileSave	TFileSaveDialog
FileOpen	TFileOpenDialog
Find String	TFindDialog
Input for user	TInputDialog
Printer abort dialog	TPrinterAbortDlg
Printer Control	TPrint Dialog
Replace String	TReplace Dialog

一、使用输入对话框

输入对话框是一种非常简单的对话框，它提示用户输入一单行文本。它即可以作为有模式对话框运行，也可作为无模式对话框来运行，在一般情况下，它是作为有模式对话框运行的。该对话框有自己的对话框资源，包存在 OWL 目录下的 Inputdia. RC 文件中，用户资源文件必须包含对话框资源文件 inputdia. rC。

创建一个输入对话框对象时，需要指定一个指向父窗口对象的指针、标题、提示、文本缓冲区及其尺寸。文本缓冲区内的信息是缺省的输入文本。当用选择了"OK"按钮或按下了回车键，输入的文本行自动传输进预先指定的字符数组，如下例所示：

```
char patientName[33]="" ;
TInputDialog(this,"Patient name"
                    "Enter the paticnt's name:"
                    patientName,sizeof(patientName)).Execute();
```

在本例中，patientName 是一文本缓冲区，当用户按下"OK"按钮时，存储用户输入的信息。PatientName 被初始化为一空的字符串作为缺省的文本。

二、应用通用对话框

作为控制的通用对话框封装了 Windows 通用对话框的功能，借助于它，可进行颜色选择，字体设置、文件名输入、寻找字符串、替换字符串、打印设置等、用户能用相似的方法构造、创建、执行、析构它们。下面将分别介绍每种对话框的具体功能。

1. 构造通用对话框

每一个通用对话框都有一个叫作 TData 的类。TData 类包含有对每一个对话框都十分

有用的数据。例如,TChooseColor 类的 TData 包含有成员数据保存选择的颜色值,以及一个保存一组用户定义的颜色的数组。下表列出了两个对所有包含 TData 类的类都通用的成员:

表2.8　通用对话框的 TData 成员

名　字	类　型	功能描述
Flags	DWORD	控制对话框的外观及动作,例如,CC_SHOWHELP 是一个标志,通知颜色选择对话框显示一个帮助按钮,用户可按下此按钮获取帮助。
Error	DWORD	这是一个错误代码。如果执行对话框时出错,Error 返回相应的错误代码。如果没有错误发生,Error 的值为0,当用户按下"Cancel"按钮,或者出现错误,Execute 函数的返回值为 IDCANCEL。所以通过检查 Error 的值,可以确定是否真的有错误发生。

每一个对话框类的构造函数都带有一个指向父窗口对象的指针,一个对 TData 宿主类对象的引用,以及若干可选的参数,如自定义的对话框模板,标题字符串、模块指针等。

下面是一个创建通用颜色选择对话框的部分代码:

```
TChooseColorDialog::TData colors;
Static TColor custColors[16]={
    0x010101L, 0x101010L,0x202020L,0x303030L,
    0x404040L,0x505050L,0x606060L,0x707070L,
    0x808080L,0x909090L,0xA0A0A0L,0xB0B0B0L,
    0xC0C0C0L,0xD0D0D0L,0xE0E0E0l,0xF0F0F0L,
};
colors.Flags=CC_RGBINIT;
colors.Color=TColor::Black;
colors.CustColors=custColors;
if(TChooseColorDialo(this,colors).Execute()==IDOK)
    SetColor(colors.Color);
```

在颜色选择框中,用户一旦选择了一种颜色,并按下了"OK"按钮,新选择的颜色将保存到 TData 对象的数据成员 Color 中。

2. 执行通用对话框

一旦构造了一个通用对话框对象,下面的任务就是执行它(若是有模式对话框),或者创建它(若是无模式对话框)。下表列出了各个通用对话框的模式类型,即有模式还是无模式:

表2.9　诸通用对话框的类型

类　型	模式类型	运　行
Color	Model	Execute
Font	Modal	Execute
Fileopen	Modal	Excute

(续)表2.9

类 型	模式类型	运 行
FileSave	Modal	Excute
Find	Modeless	Create
Find/replace	Modeless	create
Printer	Modal	Execute

执行上述对话框时，应检查 Execute 函数的返回值，确定用户是按了"OK"按钮，还是"Cancel"按钮，或者是有错误发生。

```
TChooseColorDialog::TData colors;
TChooseColorDialog colorDlg(this,colors);
if(colorDlg.Execute()==IDOK)
  // OK:data.Color==the color the user chose
else if (data.Error)
  // error occurred.
  MessageBox("Error in color dialog box!",GetApplication()->
             Name,MB_OK|MB_ICONSTOP);
```

3. 应用颜色通用对话框

该对话框为用户提供了一种在应用程序运行时选择颜色的方法。例如在一个绘图程序中选择油漆桶的颜色。

在构造一个对话框对象时，TChooseColorDialog::TData 有几个成员需要初始化。

表2.10 颜色通用对话框的成员

TData 成员	类 型	功能描述
Color	TColor	已选择的颜色。当用户执行对话框时，它是缺省的颜色；当用户关闭对话框时，它是选择的颜色。
CustColors	TColor *	一个指向包含十六个用户自定义颜色数组的指针。输入时，它表示缺省的自定义颜色；输出时，表示已选择的自定义颜色。

在下面的例子中，创建一个颜色通用对话框，为窗口对象的数据成员 Color 选择颜色值。然后利用 Color 的值，调用 TWindow::Invalidate 函数重绘窗口为选择的颜色。

```
void
TCommDlgWnd::CmColor()
{
    TChooseColorDialog::TData choose;
    static TColor custColors[16]={
        0x010101L,0x101010L,0x202020L,0x303030L,
        0x404040L,0x505050L,0x606060L,0x707070L,
        0x808080L,0X909090L,0xA0A0A0L,0xB0B0B0L,
```

```
        0xC0C0C0L,0xD0D0D0L,0xE0E0E0L,0xF0F0f0L
    };
    choose.Flags=CC_RGBINIT;
    choose.Color=Color;
    choose.CustColors=custColors;
    if(TChooseColorDialog(this,choose).Execute()==IDOK) {
        Color=choose.Color;
    }
    Invalidate();
}
```

4. 应用字体通用对话框

该种框用于在应用程序中选择字体,包括字体的字形,尺寸及字体网格等。例如在字处理程序中,可能会用该对话框为某一段文章选择字体。

在创建对话框之前,TChooseFontDialog::TData 有几个数据成员需要初始化:

表2.11 字体通用对话框 TData 数据成员

TData 成员	类　型	功能描述
DC	HDC	打印机的设备上下文句柄。必须设置 Flags 为 CF_PRINTERFONTS,其他设置均被忽略。
LogFont	LOGFONT	一个 LOGFONT 对象的句柄。LOGFONT 对象定制字体的外观。若用户指定了 CF_INITTOLOGFONTSTRUCT 标志,执行对话框时,对话框显示时具有缺省的指定字体。关闭对话框时,LogFont 保存有用户选择的字体。
PointSize	int	所选字体的点的大小。输入对,保存的为缺省值;输出时,为用户选择的值。
color	TColor	当设置了 CF_EFFECTS 时,为选择字体的颜色。
Style	char far *	对话框风格。
FontType	WORD	一组描述所选字体风格的标志。
SizeMin	int	最小值和最大值。
SizeMax	int	如果设置了 CF_LIMISIZE 标志,则为用户能够选择的点的尺寸。

在下面的例子中,创建了一个字体通用对话框为父窗口选择字体,选择的字体保存在父窗口对象的数据成员 Font 中。然后按照选择的字体在窗口中输出文本。注意在本例中是怎样应用 TFont 创建字体对象的。

```
void
TCommDlgWnd::CmFont()
{ TChooseFontDialog::TData FontData;
    FontData.DC=0
    FontData.Flags=CF_EFFECTS | CF_FORCEFONTEXIST | CF_SCREENFONTS;
    FontData.Color=Color;
    FontData.Style=0;
```

```
    FontData.FontType=SCREEN_FONTTYPE;
    FontData.SizeMin=0;
    FontData.SizeMax=0;
    if (TChooseFontDialog(this,FontData),Execute()==IDOK)
    {
        delete Font;
        Color=FontData.Color;
        Font=new TFont(&FontData.LogFont);
    }
    Invalidate();
}
```

5. 应用文件打开通用对话框

该对话框在应用程序中取代了多种不同类型的对话框，主要用于文件打开。

在创建对话框对象之前，TOpenSaveDialog::TData 有几个成员需要初始化。初始化的方法有两种，一种是直接赋值，另一种是利用 TOpenSaveDialog::TData 的构造函数，该构造函数带五个参数，这五个参数是：Flags，Filter，CustomFilter，InitialDir 和 DefExt. 这五个参数的缺省值均为零。

表2.12　文件打开存储通用对话框 TData 的数据成员

TData 成员	类　型	功能描述
FileName	char *	文件名，在输入时，为缺省的文件名；输出时，包含了已选择的文件。
Filter	char *	文件名过滤器及过滤模式。写法如下：fiter/fiter pattern/.... 这里，过滤器为 DOS 下的文件通配符、过滤器可有多个，中间用|隔开。
CustomFilter	char *	用户自定义的过滤器。
FileterIndex	int	作为缺少值显示的过滤器的索引。
CustomFilter	char *	打开文件对话框时显示的文件目录，若为零，则显示当前目录。
DefExt	char *	若 FileName 没有扩展名，则 DefExt 加到 FileName 之后作为文件的扩展名。若 DefExt 为零，则不进行上述操作。

在下面的例子中，创建一个文件打开对话框提示，用户输入文件名，如果出现错误(Execute 返回 IDCANCEL，Error 返回一个非零值)，将弹出一个消息框：

```
void
TCommDlgWnd::CmfileOpen()
{ TFileOpenDialog::TData FilenameData
        (OFN_FILEMUSTEXIST|OFN_HIDEREADONLY|OFN_PATHMUSTEXIST,
                "All Files(*.*)|*.*|Text Files (*.txt)|*.txt|",
                0,"","*");
    if (TFileOpenDialog(this,FilenameData).Execute()!=IDOK){
        if (FilenameData.Error!=0) {   // 0 value means user selected Cancel
        char msg[50];
        sprintf(msg,"GetOpenFileName returned Error #%ld",FilenameData.Error);
```

```
            MessageBox(msg,"WARNING",MB_OK|MB_ICONSTOP);
        }
    }
}
```

6. 应用文件保存通用对话框

该对话框为用户提供了一个选择文件的途径。它可以替代前面介绍的应用程序中使用的多种不同类型的对话框。

TOpenSaveDialog::TData 类在文件打开对话框中已经用到,在文件保存对话框中仍然会用到。

在下面的例子中,创建一个文件保存对话框,提示用户输入一个文件名,缺省的目录为\Windows,缺省的文件名后缀为.BMP。

```
void TCunrasWindow:: CmFileSave As()
{ TOpen SaveDialog::TData data
    OFN_HIDEREADONLY|OFN_OVERWRITEPROMPT,
    "Bit map Files (*.BMP)|*.bmp|",
    0,
    "\\Windows","BMP");
    if TFileSaveDialog(this ,data).Execute()==IDOK){
    // Save Data to file
    ifstream is(FileData->fileName);
    if( ! is)
    MessageBox("Unable to open file","File "error",
    MB_OK|MB_ICONEXCLAMATION);
    else
    // Do file output
    }
}
```

7. 应用寻找/替换通用对话框

该对话框允许用户寻找并有选择地替换应用程序中的数据。它具有很强的灵活性,不仅可以处理文本数据,还可以处理数据库中的数据。使用该对话框的最简单方法是借助于控制类 TEditSearch 和 TEditFile;它补充了编辑控制的功能。使用户能够在编辑控制中进行寻找与替换操作。如果处理的是基于文本的数据,用户可以手工进行寻找与替换。

(一)构造与创建寻找/替换对话框

由于寻换/替换对话框是无模式的,故需要在父窗口对象中定义一个指向它的指针,以此实现与对话框的信息传输。

由于是无模式的对话框,所以应以响应命令的方式构造与创建寻找/替换对话框。(如菜单中的选项 Search/Find,Search|Replace)。然后显示之,提示用户输入欲寻找的信息。

TFindReplaceDialog::TData 具有标准的 Flags 成员,以及保存寻找与替换信息的数据

成员。

在下面的例子中，在父窗口对象中定义了一个指向寻找/替换对话框的指针，以及构造与创建寻找/替换对话框的命令事件响应函数。

```
class TDatabaseWindow:Public TFrameWindow
{
    ...
    TFindReplaceDialog::TData SearchData;
    TFindReplaceDialog * SearchDialog;
    ...
};
void TDatabasewindow::CmEditFind()
{ if (! SerarchDialog){
        SearchData.Flags|=FR_DOWN;
        SearchDialog=new TFindDialog(this,SearchData);
        SearchDialog->Create();
    }
}
```

（二）处理寻找/替换信息

由于寻找/替换对话框是无模式的，它们通过应用 Windows 注册的消息 FINDMSGSTRING 与父窗口对象进行通讯，故需要编写一个响应上述消息的函数，该函数带两个参数：一个 WPARAM 和一个 LPARAM。返回一个 LRESULT 值。其中的 LPARAM 参数包含一个指针，该指针必须传给对话框对象的 UpdateData 函数。

调用 UpdataData 函数之后，必须检查标志 FR_DIALOGTERM，当用户关闭对话框时，对话框的代码将设置该标志。另外还必须将指向对话框对象的指针置零，表明该指针已经无效。在以后使用时，需要重新构造与创建对话框对象。

只有在 FR_DIALOGTERM 标志没有被设置的情况下才能响应 FINDMSGSTRING 消息，进行实际的信息寻找操作。这可能像编辑控制的 Search 函数一样简单，也可能像替换 Paradox 或 dBASE 数据库一样复杂。

在下述例子中，EvFindMsg 是事件响应函数，对一个已注册的消息做出响应。它首先调用 UpdateData 函数，然后检查标志 FR_DIALOGTERM，如该标志没有被置位，则调用另外一个函数执行信息寻找。

```
DEFINE_RESPONSE_TABLELCTDatabaseWindow,TFramWindows)
...
EV_REGISTERED(FINDMSGSTRING,EvFindMsg),
END_REPONSE_TABLE;
:
LRESULT TDatabaseWindow::EvFindMsg(WPARAM,LPARAM lParam)
{ if(SearchDialog) {
  SearchDialog->UpdataData(lParam);
```

```
if(SearchData·Flags & FR_DIALOGTERM) {
  SearchDialog=0;
  searchDialog=0;
  SearchCmd=0;
  } else
      DoSearch();
}return 0
}
```

(三)处理 FindNext(寻找下一个)命令

当寻找/替换对话框可见时,它有一个"Find Next"按钮可供用户使用。大多数应用程序在 Search 菜单中也支持"Find Next"命令。因此用户可利用该命令,而不必打开寻找/替换对话框,然后按"Find Next"按钮。利用 TFindDialog 和 TReplace Dialog 很容易实现同样的功能。

设置 FR_FINDNEXT 标志与按下"Find Next"按钮具有同样的作用:

```
void TDatabaseWindow::CmEditFindNext()
{ SearchDialog->UpdataData();
  SearchData. Flags|=FR_FINDNEXT;
  DOSearch();
}
```

8. 应用打印机通用对话框

有两种该类型的对话框:Print job 对话框和 Print Setup 对话框。利用 Print job 对话框,可以选择打印内容、打印位置、打印数量、打印份数等。利用 Print Setup 对话框从已安装的打印机中选择打印机、纸张来源、纸张大小、以及输出方向等。

利用 TPrintDialog::TData 的成员可以控制该种类型的对话框的外表及动作:

表2.13 打印机通用对话框 TData 成员

TData 成员	类 型	功能描述
FromPage	int	输出的第一页,如果设置了 PD_PAGENUMS 标志,在输入时,指定缺省的第一页;输出时,指定用户选择的第一页。
ToPage	int	输出的最后一页。如果设置了 PD_PAGNUMS,在输入时,指定缺省的最后一页;在输出时,指定用户选择的最后一页。
MinPage	int	用户所能选择的可打印纸张的最小值。
MaxPage	int	用户所能选择的可打印纸张的最大值。
Copies	int	打印份数。输入时,为缺省的打印份数;输出时,为用户选择的打印份数。

在下面的例子中,CmFilePrint 执行一个标准的 Print job 对话框,利用 TPrintDialog::TData 确定打印内容。在 CmFilePrintSetup 函数中,加入了一个标志 PD_PRINTSETUP,用以自动启动 print setup 对话框:

```
void TCanvas::CmFilePrint()
```

```
{ if (TPrintDialog (this,data). Execute()=IDOK)
  // use TPrinter and TPrinterOut to print the Drawing
}
void TCanvas::CmFilePrintSetup()
{ static TPrintDialog::TData data;
  data. Flags|=PD_PRINTSETUP;
  if(TPrintDialog(this,data,0),Execute()==IDOK)
  // print
}
```

2.5 控制对象

Windows 提供了一系列控制,这些控制是带有特殊动作的用户接口元素。ObjectWindows 提供了几种用户定义控制,为这些控制提供了界面对象,所以可以在应用程序中使用它们。控制的界面对象也称为控制对象。

本节包含以下内容:

■ 所有控制对象的公共行为;
  - 构造和撤除控制对象。
  - 与控制对象通信。

■ 使用不同的控制对象

■ 设置和读取控制值

2.5.1 控制类

下表是 ObjectWindows 提供的所有控制类:

Control	Classname	描述
标准 Windows 控制		
List box	TListBox	项目选择列表框
Scrollbar	TScrollBar	滚动条(如滚动窗口和列表框中的滚动)带有方向箭头机滑片
Button	TButton	带有文本的按钮
check box	TcheckBox	拾取框。带文本说明用于表示当前某个选择是设置的还是清除的
Radio button	TRadioButton	互锁按钮,用于一组选择中互斥的选择
Group box	TGroupBox	一个静态矩形框,左上角有选择文本
Edit antrol	TEdit	供用户输入文本的域
Staric antrol	TStatic	用户不能修改的可见文本
Combo box	TComboBox	组合框

(续)

Control	Classname	描述
Objectwindows 控制		
Slider	THSlider 和 TVslider	水平和垂直控制，让用户在 一个从高到低的范围里选择
Gauge	TGauge	静态控制，用于显示处理的范围值

TGauge 是 Borlandc++4.0新增的控制类

什么是控制对象

相对 Windows，控制是一类特殊的窗口，在 ObjectWindows 中，TControl 从 TWindow 中派生，控制对象和窗口对象在作为子窗口时，具有相同的处理行为。但标准控制和其他窗口的区别在于：窗口要处理自己的消息，并负责刷新。ObjectWindows 控制自己来完成这些任务，因为 ObjectWindows 控制类中包含了重绘控制和处理事件的代码。

很多情况下，可以直接用前表中列出的这些类的实例，但有时你可能要创建这些类的派生类以完成特殊的功能。如：可能要从 TListBox 列表框类中派生出 TFontListBox 类，用这个类来列出应用程序能够获得的所有字体，并在创建 TFontListBox 类的实体时自动显示这些字体。

2.5.2 构造和撤除控制对象

不管使用什么种类的控制对象，对每种控制对象都必须完成几项任务：

- 构造控制对象
- 显示控制
- 撤销控制

一、控制对象的构造

构造控制对象和构造任何其他子窗口没有什么区别。一般，父窗口的构造函数调用所有子窗口的构造函数。控制和父窗口用特殊的方式通信(称通知消息)。

为构造和初始化一个控制对象：

1. 在父窗口中增加一个控制对象指针数据成员。
2. 调用控制对象的构造函数。
3. 修改任何控制属性。
4. 在 SetupWindow 中初始化控制。

以下详细讨论每一书。

(一)增加控制对象指针数据成员

通常，在窗口中构造一个控制时，要在窗口对象中保留一个指向控制的指针。这在获取控制的成员函数时会很方便。下面是一个父窗口对象的框架。在框架中声明一个指向按钮控制对象的指针。

```
Class TMyWindows:Public TWindow{
    TButton * OKButton;
    ...
};
```

一些不用操作的控制如静态文本和成组框，不用这些指针数据成员。下列创建一个成组框和一个按钮，成组框在窗口中没有指针数据成员，而按钮在窗口中有指针数据成员。

```
TMyWindow::TMyWindow(TWindow * parent,const char for * title)
           :TWindow(parent,title)
{ new TGroupBox(this,ID_GROUPBOX,"Group box",10,10,100,100);
  OkButton=new TButton(this,IDOK,"OK",10,200,50,50,TRUE);
}
```

(二)调用控制对象的构造函数

一些控制对象的构造函数要传递参数规定控制对象的特性。这些参数包括：

- 指向父窗口对象的指针
- 一个资源标识符
- 左上角的 X 坐标
- 左上角的 Y 坐标
- 宽度
- 高度
- 库 ID 值(可选)

如 TListBox 的构造函数定义如下：

```
TListBox(TWindow * parent,int,id,int x,int y,int w,int h,
         TLibId LibId=0);
```

将控制对象和界面元素(如对话框)联系起来的构造函数是：

```
TListBox(TWindow *parent,int resourceId,
         TModule *Module=0);
```

(三)修改控制属性

所有控制对象具有窗口风格 WS_CHILD，WS_VISIBLE，WS_GROUP，和 WS_TABSTOP，如果要改变属性，只要操作 Attr.Style，这在第六节中讲述，每种控制还有其他一些风格，来定义特殊的属性。

每种控制从基类中继承了一些窗口风格。一般很少给 Attr.Style 赋值、相反，应该用操作符|=和&=来屏蔽一些窗口风格。如：

```
// mask in the WS_BOKDER Window style
Attr.Style|=WS_BORDER;
// Mask out the WS_VSCROLL Style
```

```
Attr.Style &=WS_VSCROLL;
```

使用以上两种赋值操作符能保证不会由于不小心而误删了一些风格。

(四)初始化控制

控制对象的界面元素在 SetupWindow 中自动创建。SetupWindow 函数成员是从父类对象中继承而来的。一定要保证,在派生一个新的窗口类时在操作其中的控制前,要先调用基类的 SetupWindow 函数。

不要在你窗口对象的构造函数中初始化控制,因为在构造函数中,控制的界面元素还没有生成。

下面是一个典型的 SetupWindow 函数:

```
void TMyWindow::SetupWindow()
{ TWindow::SetupWindow();   // Creat child controls
  list1->Addstring("Item 1");
  list1->AddString("Item 2");
}
```

二、显示控制

没有必要调用窗口函数 Show 来显示窗口,控制是子窗口,Windows 自动显示和刷新它们。但是;根据需要可以用 Show 来隐藏或显示控制。

三、撤销控制

父窗口负责撤销控制。控制的界面元素自动随父窗口的消失而消失。当用户关闭窗口或应用程序时,父窗口的析构函数自动撤除它的子窗口对象(包括控制对象)。

2.5.3 和控制对象的通信

窗口对象与其控制之间的通信和对话框与共控制之间的通信在某种程序上比较相似。像对话框一样,窗口需要有一种机制来操纵其控制,响应控制事件,如列表框选择。

一、操纵控制

对话框操纵其控制的一种方法是通过从 TWindow(看第二节)继承来的成员函数向控制发送消息,像控制消息 LB_ADDSTRING。控制对象极大地简化了这个过程,提供了一些成员函数来向控制发送消息。例如,TListBox::AddString 带一个参数,通过调用列表框对象的 HandleMessage 函数将参数加入列表框。

```
TListBox::AddString(const char far * str)
{ return(int HandleMessage (LB_ADDSTRING,0,(LPARAM)str);
}
```

通过指针来调用控制对象的成员函数:

```
Listbox1->Addstring ("Atlantic City");
```

二、响应控制

当用户和控制发生时，Windows 会发送几种控制消息，如何响应控制消息，参见第六节。

三、使窗口像对话框一样工作

对话框让用户用 Tab 键在所有对话框控制之间循环选择。用户还能用箭头键在组合框中选择 radio 按钮。如果要为 Windows 的控制模拟这种键盘接口，调用 EnableKBHandler(在窗口对象的构造函数中调用)。

2.5.4 使用特殊的控制

每一种类型的控制和其他类型的控制总会有些区别。这一部分讨论如何使用标准 Windows 控制和 ObjectWindows 提供的控制。

一、使用列表框控制

列表框是供用户选择项目的最简单的方法。TListBox 封装了列表框。出于各自的目的，TListBox 定义了一些成员函数。

- 创建列表框
- 修改列表项
- 查询列表项
- 找出用户选择的项

(一)构造列表框对象

TListBox 的构造函数之一带七个参数：父窗口指针，资源标识符，控制的 x，y，h，w 尺寸和可选的库标识符。

```
TListBox(TWindow *parent,int resourceId,int x,int y,
        int w,int h, TLibId LibId=0);
```

TListBox 获取缺省的控制风格，并增加 LBS_STANDARD 属性，这个属性是 LBS_NOTIFY(接收通知消息)，WS_VSCROLL(具有垂直滚动条)LBS_SORT(按字母顺序排序列表框)WS_BORDER(有一个边框)的组合风格，如果想用不同的列表框风格，可以在列表框的构造函数或父窗口的构造函数中修改 Attr.Style，例如，创建一个列表框，不需要排序，代码如下：

```
ListBox=new TListBox(this,ID_LISTBOX,20,20,340,100);
ListBox->Attr·Style &=LBS_SORT;
```

(二)修改列表框

在创建列表框之后，需要用列表项目来填充(必须是字符串)，然后就可以增加，插入、删除或清除列表框。下表是可以使用的成员函数的总结表：

成员函数	功能
ClearList	删除一切列表项
DirectoryList	在列表框中放置文件名
AddString	增加一项
InsertString	插入一项
DeleteString	删除一项
SetSelIndex,SetSel,SetSelString	选择一项
SetSelStrings,SetSelIndexes,SetSelItemRange	选择多项
SetTopIndex	滚动列表框,以便指定项可见
SetTabStops	为多列列表框设置 tab
SetHorizontalExtent	设置列表框水平滚动的像素值
SetColumnWidth	设置多到列表框的宽度
SetCaretIndex	设置当前聚焦项的索引值
SetItemData	设置与指定索引项项目相关的 DWORD 值
SetItemHeight	设置指定索引项或所有项的高度

(三)查询列表框

查询列表框或列表框选项有关信息的函数如下表:

成员函数	功能描述
GetCount	返回列表框中的个数
FindString 或 FindExactString	查找字符串
GetTopIndex	获取列表框顶部选项的索引
GetCaretIndex	获取当前聚焦项的索引
GetHorizontalExtent	获取列表框能滚动的水平像素值
GetItemData	由 SetItemData 设置的 DWORD 数据
GetItemHeight	获取指定选项的像素高度
GetItemRect	获取指定选项的显示矩形
GetSelCount	获取所有选项的个数
GetSetIndex 或 GetSel	获取被选项的索引
GetSelString	获取被选项(一项)
GetSelStrings 或 GetSelIndexes	获取被选项(多项)
GetString	获取指定索引的选项
GetStringLen	获取指定项的字串长度

(四)响应列表框

列表框的修改和查询成员函数可以用来设定值,查询控制的状态,然而,在运行时,要想知道用户对列表框做了什么操作,必须响应控制传来的通知消息。

用户只能操作列表框的以下项:浏览列表项;单击点取某项;双击点取某项,当用户做以下操作之后,Windows 就会发送一个列表框通知消息给列表框的父窗口。通常要在父窗口对象中定义通知消息响应成员函数来处理父控制的通知。

下表列出了常用的列表框通知消息

文件响应表宏定义	描述
EV_LBN_SELCHANGE	通过单击鼠标选取了某项
EV_LBN_DBLCLK	通过双击鼠标选取了某项
EV_LBN_SELCANCEL	用户取消了某项选择
EV_LBN_SETFOCUS	用户通过单击或双击鼠标或 tab 键聚焦某项
EV_LBN_KILLFOCUS	用户按了 Tab 键或选取了其他控制，因而使当前选项失去聚焦

下列是父窗口对象处理 LBN_SELCHANGE 通知消息：

```
DEFINE_RESPONSE.TABLE1(TLBOXWindow,TFrameWindow)
EV_LBN_SELCHANGE(ID_LISTBOX,EvListBoxSelChange),
END_RESPONSE_TABLE;
void TListBoxWindow::EvListBoxSelChange ( )
{ int index=ListBox->GetSelIndex();
    if (ListBox->GetStringLen(index)<10){
        char string[10];
        ListBox->GetSelString(String,Sizeof(string));
        MessageBox(String,"You Selected;",MB_OK);
    }
}
```

二、使用静态控制

静态控制是不能更改的文本单位或简单的图形，尽管应用程序可以修改静态控制文本，用户一般不用和静态控制打交道。

EXAMPLE\OWL\OWLAPI\STATIC 目录下有示例。

(一)构造静态控制对象

由于用户不能和静态控制直接打交道，所以应用程序不需要从静态控制接收控制通知消息，因而，创建静态控制时可以用-1作为控制的 ID 值。如果想用 TWindow::SendDlgItemMessage 操纵静态控制，必须要有一个 ID 值。

TStatic 构造函数之一如下：

```
TStatic(TWindow * parent,int resourceId,const char far
        * title,int x,int y,int w, int h, UINT textLen, TLibId LibId=0);
```

这个构造函数带九个参数，其中七个是公共的(父窗口指针，资源 ID，控制的 x，y，w，h 尺寸；可选的库 ID 号)，另外两个特殊的参数是：静态控制欲显示字符串及最大长度(包括终止号 NULL)。一个典型的静态控制的构造函数如下：

```
new TStatic(this,-1,"Sample & Text",70,20,200,24,0);
```

如果要想改变静态控制的文本，必须要在父窗口对象的构造函数中保留一个数据成员，

这样，才能调用静态控制的成员函数。如果不必要改变静态控制文本，就没有必要保留一个数据成员。

静态控制使用缺省控制风格，增加了 SS_LEFT(文本左对齐)，删去了 WS_TABSTOP 风格(阻止用户用 Tab 选择)在静态控制构造函数中修改 Attr. Style。如下列代码将文本居中放置。

```
Attr.Style=(Attr.Style & ~SS_LEFT)|SS_CENTER;
```

为了显示控制的助记符，可以在静态控制文本串的一个或多个字下划一条线，只要在字串里想划线的字的前面插入一个 & 字符，例如，在 Text 的下划线，用 &Text。如果要用 & 符号，使用 SS_NOPREFIX 静态风格。

(二)修改静态控制

TStatic 提供两个函数来修改静态控制的文本：SetText 和 Clear。SetText 用传递的字符串参数设置文本，而 Clear 则删去文本。不能改变 SS_SIMPLE 风格创建的静态控制的文本。

(三)查询静态控制

TStatic::GetTextLen 返回静态控制的文本长度，而 TStatic 的 GetText 函数获取字串本身。

三、使用按钮

按钮(有时称命令按钮)，当按下一次时，它便执行一次任务。有两类按钮：缺省的和非缺省的。缺省钮(具有 BS_DEFPUSHBUTTON 风格)，具有醒目的边框，以显示对用户的缺省响应，非缺省按钮具有 BS_PUSHBUTTON，参见 Excmple\OWL\BWCC 下的 Button 控制例子。

(一)构造按钮

TButton 的构造函数带九个参数：指向父窗口的指针，资源 ID，控制的 x、y、h、w 尺寸，可选的标识符；另外的参数是文本字串，用于规定按钮标签；一个 BOOL 标志，用于标识按钮是否是缺省按钮。

构造函数的声明如下：

```
TButton(TWindow * parent,int resource Id,const char far
        * text int x,int y, int w,int H,BOOL isDefault=FALSE,TLibId libId=0);
```

典型的按钮构造函数如下：

```
btn=new TButton(this,ID_BUTTON,"DO_IT",38,48,316,24,TRUE);
```

(二)响应按钮

当用户按下按钮时，按钮的父窗口接受到一个通知消息如果父窗口解释消息，它就可以显示一个对话框，保存文件等来响应这些事件。

为了解释和响应按钮事件，必须给按钮定义一个命令响应成员函数。下例用 ID 值 ID_BUTTON 来处理用户单击按钮的响应。

```
DEFINE_RESPONSE_TABLE1(TTestWindow,TFrameWindow)
EV_COMMAND(ID_BUTTON,HandleButtonMsg),
END_RESPONSE_TABLE;
void TTestWindow::Handle ButtonMsg()
{ // Button was pressed
}
```

四、使用拾取框和互锁按钮控制

拾取框一般代表两种状态的选项。用户可以拾取或不拾取控制，或者忽略控制，在成组拾取框中，可能要拾取一个或所有的项。例如，在应用程序中，可能用拾取框来使声音有效或无效。

Radio 按钮，用来在几个互斥的选项中拾取一项。例如，用 radio 按钮在一系列声音中选取一种。

TCheckBox 从 TButton 派生出来，由于 radio 按钮和拾取框共用一些行为，所以 TRadioButton 从 TCheckBox 派生。

拾取框和 radio 按钮有时称为选取框。当在屏幕上显示时，选取框要么是选择的要么是不选择的。当用户按下选取框时，这就是一个事件，会产生一个 Windows 通知消息。正如其他控制一样，选取框的父窗口通常解释和响应这些通知消息。

参见 EXAMPLE\OWL\OWLAPI\BUTTON 下的 radio 按钮和拾取框控制例子。

(一)构造拾取框和 radio 按钮

TCheckBox 和 TRadioButton 构造函数带九个参数；父窗口指针，资源标识 ID，控制的 x,y,w,h 尺寸，可选的库 ID；另外的参数是文本字串和一个指向成组框对象的指针。该成线框将选择组合在一起。如果成组框对象指针为0。则选择框就不是成组框的所属部分。下列是构造函数。

```
TCheckBox(TWindow *parent,int resourcedId,const char far *title,
          int x, int y, int w,int h, TGroupBox * group, TLibId libId=0)
TRadioButton(TWindow *parent,int resourceId, const char far * title,
             int x,int y, int w ,int h, TGroupBox * group,TLibId LibId=0);
```

下例是选择框的典型构造函数：

```
checkBox=new TCheckBox(this,ID_CHECKBOX,"Check Box",158,
                       12,150,26,0);
GroupBox=new TGroupBox(this ,ID_GROUPBOX,"Group Box",158,
                       102,176,108);
RButton1=new TRadioButton(this,ID_RBUTTON1,"Radio Button1",
                          174,128,138,24,GroupBox);
RButton2=new TRadioButton(this,ID_RBUTTON2",Radio Button2",
                          174,162,138,24,GroupBox);
```

拾取框具有缺省风格 BS_AUTOCHECKBOX,它表明 Windows 通过开/关拾取框来处理对拾取框的鼠标击发动作。如果没有该风格,就不能设置拾取框的状态,radio 按钮缺省的风格是 BS_AUTORADIOBUTTON,它表明 Windows 处理 radio 按钮时,拾取组中的一个 radio 按钮,同时不拾取组中的另一个按钮。如果没有 BS_AUTORADIOBUTTON,用户必须解释 radio 按钮的通知消息。

(二)修改选取框

选取和不选取选取框是应用程序的用户的工作,而不是程序的任务,但在有些情况下;你的应用程序需要控制选取框的状态。例如,如果用户打开一个文本文件,你可能想自动检测标有"Save As ANSI text"的拾取框,TCheckBox 定义了几个函数来修改拾取框的状态。

成员函数	功能描述
check 或 SetCheck(BF_CHECKED)	设拾取状态
UnCheck 或 SetCheck(BF_UNCHEKED)	设不拾取状态
Toggle	在状态之间切换
SetState	设置状态
SetStyle	设置按钮风格

当使用 radio 按钮的这些成员函数时,ObjectWindows 保证每个组中有一个 Radio 按钮被选中(如果这些 Radio 按钮被指定为一组的话)。

(三)查询选取框

查询一个选取框是寻找和响应其状态的一种方式,Radio 按钮有两种状态;选中(BF_CHECKED)和未选中(BF_UNCHECKED)拾取框还有另外一种状态(可选);变灰(BF_GRAYED)下表列出了选取框的查询函数。

成员函数	功能描述
GetCheck	返回检测状态
GetState	返回检测、高亮、或聚焦状态

五、使用成组框

成组框最简单的形式,就是一个标有静态文本的矩形,在这个矩形中,将其他控制组合在一起。

(一)构造成组框

成组框构造函数带有八个参数,其中七个参数在控制对象构造函数中很常见(父窗口,资源标识符,控制的 x,y,h,w 尺寸和一个可选的库标识符),另外一个参数是用来标识成组框的文本字串参数。

```
TGroupBox(TWindow * parent,int resourceId, const
          char far * text, int X, int Y, int W,int H, TLibId LibId=0);
```

（二）组织成组框其中的控制

通常成组框是联系一组其他控制的；然而，从逻辑上讲，它也可以联系一组选取框（拾取框和 radio 按钮）。这个逻辑组完成自动的未选取功能(BS_AUTOCHECK,BS_AUTODARIOBUTTON)。

要增加一个选取框到成组框中，只要在选取框的构造函数调用中传递一个指针给成组框对象。

（三）响应成组框消息

当一个事件产生时，可能改变成组框的选择（如：当用户按一个按钮或应用程序调用 Check 时），Windows 给成组框的父窗口发送一个通知消息，父窗口将这些成组框的消息作为一个整体来解释，而不是响应成组框中的单个选取框，如果要查出成组框中的那个控制受到了影响，可以读取每个控制的当前状态。

六、使用滚动条

滚动条是主要的一种工具，它用来改变应用程序窗口的用户浏览区，列表框，或一个组合框。然而，有时为完成一些特殊任务，需要一个单独的滚动条，例如用滚动条来控制温度计的温度，或绘图程序中的颜色。使用 TScrollBar 对象来创建一个单独的，用户化的滚动条。

参见 Examples\owl\owlapp\scroller 下的例子。

（一）构造滚动条

TScrollBar 带八个参数；其中七个在控制对象的构造函数中很常见（父窗口，资源标识符，控制的 x，y，h，w 参数和一个可选的库标识符），另外一个参数是一个 BOOL 标志，用来规定滚动条是水平的，还是垂直的。下面是 TScrollBar 的构造函数。

```
TScrollBar(TWindow * parent,int resourceId int X,int Y,
           int W,int h,BOOL isHScrollBar,TLibId LibId=0);
```

如果在水平滚动条中将高度设为0，或在垂直滚动条中将宽度设为0，则 Windows 规定标准的宽和高，下面代码创建一个标准高度的水平滚动条：

```
new TScrollBar(this, ID_THERMONETER,100,150,
               180,0,TRUE);
```

TScrollBar 的构造函数在创建滚动条时，指定水平滚动条具有 SBS_HORZ 风格，垂直滚动条具有 SBS_VERT 风格，另外通过修改滚动条的 Attr. Style，指定其他的风格，如 SBS_TOPALIGN。

（二）控制滚动条的范围

滚动条的一个属性是范围（range），它是一系列所有可能的 thumb 位置，thumb 是用户拖动或滚动的滑块。每一个位置都是一个整数，父窗口利用这个，整数（位置）设置和查询滚动条，缺省时，滚动条的范围是1～100。

thumb 的最小位置（垂直滚动条在顶部，水平滚动条在左边）为1，最大位置为100，使用

SetRange 设置不同的范围。

(三)滚动量的控制

滚动条还有另外两个属性:Line Magnitade 和 Page Magnitude。Line Magnitule 初始化为1,用(范围单位),当用户单击滚动条的箭头时,thumb 移动一个 Line Magnitule Page Magnitule 初始值为1(用范围单位),当用户单击滚动条的区域时,thumb 移动 Page Magnitude。可通过函数 Line Magnitude 和 Page Magnitade 来修改其值。

(四)查询滚动条

查询滚动条的两个成员是:

- GetRange 获取上限和下限范围
- GetPosition 获取 thumb 的当前位置。

(五)修改滚动条

一般由用户修改滚动条,但应用程序也可以直接修改滚动条。

- SetRange 设置滚动条范围
- SetPositon 设置 thumb 的位置
- DeltaPos 移动 thumb 的位置。

(六)响应滚动条消息

当用户单击滚动条的箭头或移动 thumb 时,Windows 会向父窗口发送一个滚动条通知消息,如果要窗口响应滚动事件,响应其通知消息。

滚动条消息和其他的控制消息有些区别。它们是基于 WM_HSCROLL 和 WM_VSCROLL,而不是基于 WM_COMMAND 命令消息,所以,要响应滚动条的通知消息,必须定义 EvHScroll 和 EvVscroll 事件响应函数,这取决于是水平滚动条还是垂直滚动条。

```
class TTestWindow:public TFrameWindow{
public:
    TTestWindow(TWindow * parent, const char * title);
    virtual void SetupWindow();
    void EvHScroll(UINT code,UINT POS,HWND Wnd);
    DECLARE_RESPONSE_TABLE(TTestWindow);
};
DEFINE_RESPONSE_TABLE1(TTestWindow,TFrameWindows)
    EV_WM_HSCROLL;
END_RESPONSE_TABLE;
```

通常,通过跟踪 thumb 的当前位置,可以响应所有的滚动条通知消息。此时,可以忽略通知代码:

```
void TTestWindow::EvHScroll (UINT code,UINT pos, HWND wnd)
{ TFramWindow::EvHScroll();   // 响应缺省的 WM_HSCROLL
    int newpos=ScrollBar->GetPosition();
```

```
        // do some processing with newpos
    }
```

避免 thumb 跟踪消息

由于用户通常很快地拖动 thumb，会产生很多的通知消息，所以当用户拖动 thumb 时，一般不要响应滚动条通知消息，当用户停止移动 thumb 后，再响应消息会更有效。要达到此目的，屏蔽掉有 SB_THUMBTRACK 代码的通知消息。

具有特殊的滚动条行为

有时，要使滚动条对象响应自己的通知消息。TWindow 中内嵌的功能支持把滚动条通知消息分给滚动条 TWindow::EvHScroll 或 TWindow::EnVScroll 执行合适的 TScrollBar 基于通知代码的成员函数，例如：

```
class TSpecializedScrollBar:public TScrollBar {
public:
    virtual void SBTop();
}
void TSpecialized ScrollBar::SBTop() {
TscrollBar::SBTop();
::sndPlaySound ( * AT-TOP-WAV,SND_ASYN();
}
```

要保证首先调用基类的函数。这些函数将把滚动条更新到新的位置。

下表列出了通知消息和相应的 TScrollBar 成员函数：

通知消息	TScrollBar 成员函数
SBLINEUP	SBLine ↑
SB_LineDOWN	SBLine ↓
SB_PAGEUP	SBPage ↑
SB_PAGENDOWN	SBPage ↓
SB_THUMBPOSITION	SBThumbPosition
SB_THUMBTRACK	SBThumbTrack
SB_TOP	SBTop
SB_BOTTOM	SBBottom

七、使用滑动快和量具

滑块(slider)是一种特殊的滚动装置；类 TSlider 从 TScrollBar 派生出来。滑块用作不滚动定位信息。THSlider 和 TVSlider 从 TSlider 派生，以完成垂直和水平滑块的功能。

gauge 是一种控制，用于显示一个正在进行的过程的持续时间或其他信息。类 TGauge 从类 TControl 派生出来，完成 gauge 的一些功能。构造函数的一个参数决定是用水平 gauge 还是垂直 gauge。水平 gauge 常用于显示处理信息，垂直 gauge 常用于显示模拟信息。

参见 examples\owl\owlappi\slider 中有关 gauge 的例子。

八、使用编辑控制

编辑控制是交互式的静态控制。它们是一些矩形区域，充满着文本，可以修改，甚至要以

由用户或应用程序来清除。编辑控制用作数据输入域是很有用的。它们支持以下的操作。

- 用户文本输入
- 文本的动态显示(由应用程序)
- Cut,Copy 和 Paste 到 Clipboard
- 多行编辑(文本编辑器)

参见 examples\owl\owlapi\validate 有关编辑控制的例子。

(一)构造编辑控制

TEdit 的一个构造函数带有参数来初始化文本串,文本串的最大长度(包括终止符 NULL)和一个 BOOL 标志以指定是否是多行编辑控制(另外的参数是父窗口,资源标识和位置坐标)。下面是 TEdit 的构造函数。

```
TEdit(TWindow * parent,int resourcedId, const char far * text,int x,int,y,
      int w, int h ,UINT textLen, BOOL multitine=FALSE,TLibId LibId=0);
```

缺省时,编辑控制具有 ES_LEFT 风格(文本左对齐);

ES_AUTOSCROLL(自动水平滚动)和 WS_BORDER(具有可见的边框围绕着编辑控制,多行编辑还有另外的风格 ES_MULTLINE(指定多行编辑控制),ES_AUTOVSCROLL(自动垂直滚动),WS_VSCROLL(垂直滚动条)和 WS_HSCROLL(水平滚动条)。

下面是典型的编辑控制的构造函数,一个是单行控制,另一个是多行控制。

```
Edit1=new TEdit(this,ID_EDIT1,"Defaut Text",20,50,
                150,30,MAX_TEXTLEN ,FALSE);
Edit2=new TEdit(this,ID_EDIT2,"",260,50,150,
                30,MAX_TEXTLEN,TRUE);
```

(二)使用 **Clipboard** 和编辑菜单

使用 TEdit 成员函数可以在编辑控制对象和 Windows Clipboard 之间直接传送文本。如果想给用户访问这些成员函数的能力,就必须给窗口提供一个 Edit 菜单。

有命令响应成员函数,如 CmEditCopy,CmEditUndo。ObjectWindows 在父窗口的 Edit 菜单中激活这些函数来响应用户选项。

下表显示了 Clipboard 和编辑成员函数及菜单命令:

成员函数	菜单命令	功能描述
Copy	CM_EDITCOPY	拷贝文本到 Clipboard
Cut	CM_EDITCUT	剪切文本到 Clipboard
Undo	CM_EDITUNDO	取消上一次操作
Paste	CM_EDITPASTE	从 Clipboard 粘贴文本
DeleteSelection	CM_EDITDELETE	删除选择的文本
Clear	CM_EDITCLEAR	清除整个编辑控制

给包含编辑控制对象的窗口增加编辑菜单时，要给使用以上命令的窗口定义一个菜单资源，不用写其他新的成员函数。

(三)查询编辑控制

用户经常要查询编辑控制，储存其中的项以备后续使用。TEdit 有很多查询的成员函数，许多编辑控制的查询和修改成员函数返回或需要指定一个行号或一行中某个字符的位置。所有这些索引均从0开始，换句话说，第一行是零行，一行中的第一个字符是字符的零位置。

下表列出 TEdit 的查询成员函数：

成员函数	功能描述
IsModified	文本是否变化
GetText	复制所有文本
GetLine	复制一行文本
GetNumLines	获取行数
GetLineLength	获取一个给定行的长度
GetSelection	获取被选择文本的起始位置
GetSubText	获取一个范围内的文本
GetLineIndex	返回指定行之前的第一个字符之前的一个字符的字符位置
GetLineFromPos	返回指定的字符位置所在的行号
GetRect	获取格式化的矩形
GetHandle	获取内存的句柄
GetFirstVisibleLine	获取第一个可见行的位置；
GetPasswordChar	返回 passwords 中的字符
GetWordBreakProc	获取 word-breaking 例程
CanUndo	查找编辑是否能取消

在多行编辑控制中有两个额外的字符：回车符('\r')和换行符 '\n'，这两个字符用来分隔文本。

TEdit 的成员函数保持了文本的格式(当从多行文本编辑控制中返回文本时)。当把文本插回编辑控制，从剪贴板粘贴，写入文件或从打印机打印时，行断开符跟编辑控制中一样出现。当使用查询成员函数来获取一个指定数目的字符时，一定要把这两个额外的字符计算在内。

(四)修改编辑控制

许多编辑控制需要应用程序精确地替换、插入、清除或选择文本。TEdit 支持这些操作、另外还有滚动编辑控制的能力。

下表是 TEdit 是修改编辑控制的成员函数：

成员函数	功 能
Clear	删除所有文本
DeleteSelection	删除选中的文本
DeleteSubText	删除某个范围的文本
DeleteLine	删除一行文本
Insert	插入文本
Paste	从 Clipboard 中粘贴文本
SetText	替换所有文本
SetSelection	选择一个范围里的文本
Scroll	滚动文本
ClearModify	清除修改标志
Search	搜索文本
SetRect 或 SetRectNP	设置格式化矩形
FormatLines	关闭/打开软换行符
SetTabStops	设置 tab 停止符
SetHandle	设置局部内存句柄
SetPasswordChar	设置 Password
SetReadOnly	使编辑控制只读
SetWordBreakProc	设置 word_breaking 过程
EmptyUndoBuffer	清除 Undo 的缓冲区

九、使用组合框

组合框是两种控制的组合;列表框和编辑/静态控制,它提供列表框相同的功能——让用户通过单击 mouse 选择一个文本项。其编辑控制,位于列表框的顶部,提供了另一种选择机制,用户可以输入所需的文本。如果组合框的列表区域被显示出来,想选择的项就会自动选择上。TComboBox 从 TListBox 派生出来,并继承了其修改,查询和选择的成员函数。另外,TComboBox 还提供了操纵组合框列表部分的成员函数,根据要求,某些组合框可以下拉(drop down)。

参见 owlapps\owlcmd 其的例子

(一)组合框的种类

有三种组合框:simple,dropdown 和 dropdownlist。所有的组合框都会显示其编辑区域,但一些组合框显示和隐藏列表区域。下表总结了每种组合框的特性。

Style	列表框	文本必须和列表匹配
Simple	No	No
DropDown	Yes	No
DropDownlist	Yes	Yes

从用户的角度看,这几种风格的组合框有些区别:

■ Simple 组合框不能隐藏其列表区域,其编辑域功能和编辑控制一样,用户可输入和

编辑文本，且文本也不必和列表框中的某一项区域，如果文本和某一项相匹配，则相应的列表项被选中。

- dropdown 和 Simple 组合框大致相同，只有一项例外。在初始状态，其列表区是不显示的。当用户单击编辑域右边的图标时，它就会出现。dropdown 组合框不用时，它比列表框或 simple 组合框占的空间要小。
- dropdownlist 的列表域和 drop down 的列表区域一样，只在需要时才显示出来。这两种组合框的区别在于编辑区。drop down 组合框的编辑域和常规编辑控制一样。而 drop down list 编辑域只显示列表框中的文本。当编辑域中的文本和列表框中的文本匹配时，就不能再输入文本。

（二）选择组合框类型

Drop downlist 只适合于从列表框中选择文本项。如选择打印机时，你只能选择系统接受的打印机。

另一方面，drop down 组合框可以接受列表框中没有的选项。drop down 组合框的典型用途是打开或存磁盘文件。用户既可以浏览目录找出列表域中合适的文件，也可以在编辑域中输入全路径名和文件名，而不管文件名是否在列表域中出现。

（三）构造组合框

TComboBox 构造函数带的参数如下：父窗口，资源标识符，控制的 x，y，w，h 尺寸，可选的库标识符，风格和文本的最大长度参数。

```
TComboBox(TWindow * parent, int id, int x ,int y,int w,
                int h, DWORD style, WORD textLen, TlibId LibId=0);
```

所有的组合框具有以下风格；WS_CHILD，WS_VISIBLE，WS_GROUP，WS_TABSTOP，CBS_SORT（排序列表项）CBS_AUTOHSCROLL（用户可以在编辑域中输入比可见长度更多的文本），WS_VSCROLL（垂直滚动条），风格是 CBS_SIMPLE，CBS_DROPDOWN，CBS_DROPDOWNLIST 之一，文本长度是编辑域可以输入的最大字符数，下例创建一个组合框。

```
Combol=new TComoboBox(this,ID_COMB1,190,30,
                      150,100,CBS_SIMPLE,20);
Combol->Attr.Style=~CBS_SORT;
```

（四）修改组合框

TComboBox 定义了几个函数来修改组合框的列表域和编辑域。下表总结了这些函数。

由于 ComboBox 从 TListBox 派生出来，所以也可以用 TListBox 的成员函数来操纵组合框的列表区。

成员函数	功　能
SetText	替换编辑区中的所有文本
SetEditSet	选择编辑区中的文本
Clear	清除编辑区中的文本
ShowList	显示列表区
HideList	隐藏列表区
SetExtendedUI	设置扩展的组合框 UI

(五)查询组合框

TCombo Box 在 TListBox 的基础上增加了几个成员函数来查询组合框的编辑和列表区的内容。下表总结了这些成员函数：

成员函数	功　能
GetTextLen	获取编辑区的文本长度
GetText	复制编辑区的所有文本
GetEditSel	返回被选择字串从开始至结束位置
GetDroppedControlRect	获取 drop down list 的矩形区
GetDroppedState	列表区是否可见
GetExtendedUI	组合框是否有扩展的 UI

2.5.5 设置和读取控制的值

为了管理具有很多子窗口控制的复杂对话框或窗口，可能要创建一个派生类来存储和检索，对话框或窗口中控制的状态。控制的状态包括编辑控制中的文本，滚动条的位置，以及按钮是否被选择。

一、使用传输缓冲区

作为创建派生类的替代，可以用一个结构来代表对话框或窗口控制的状态。这个结构称 transter buffer。因为控制的状态从控制传向 buffer 然后从 buffer 传向控制。

例如：应用程序创建一个模式对话框，当用户选择关闭后，从传输缓冲区抽取有关控制状态的信息。如果用户想再次显示以框，就可以从传输缓冲区传送控制状态。另外还可以指定传输的方向，如将控制的状态恢复成原来的状态。可以用传输缓冲区中的数据初始化每个控制的状态。窗口或无模式对话框也可以用传输机制设置或检索状态信息。

传输机制需要使用 ObjectWindows 对象来代表欲传输数据的控制，要使用传输机制，必须做三件事：

■ 定义传输缓冲区，其中为每个要传输数据的控制设一个变量。
■ 定义相应的窗口和对话框。
■ 传输数据。

二、定义传输缓冲区

传输缓冲区是一个结构，每一个参与传输的控制都有一个成员。这些成员称为"instance variables"。窗口或对话框也有控制没有状态要传输，如：缺省时，按钮，组合框和静态控制不参与传输。

要定义一个传输缓冲区，就要为每个参与传输的窗口或对话框定义一个实例变量。没有必要为每个控制定义一个变量，只须为要传输值的控制定义变量。传输缓冲区为除开按钮成组框，静态控制外的所有控制存储变量。例如：

```
Struct TSample TransferStruct
{ Char editCtl[SizeofEditCtl];
  WORD CheckBox;
  WORD RadioButton;
  TListBoxData * listBox;
  TComboBoxDat a * comboBox;
  TScrollBarDat a * scrollBar;
}
```

每个控制存储的信息不同下表列了 ObjectWindows 控制的 transfer buffer：

控制	类型	描述
Static	char array	包括 NULL 结束符在内的最大文本长度的字符数组，缺省时，静态控制不参与传输，但可以使传输有效
Edit	char array	包括 NULL 结束符在内的最大文本比度的字符数组
Listbox	TListBoxData *	指向 TListBoxData 实例的指针
ComboBox	TComboBoxData *	指向 TComboBoxData 实例的指针。TComboBoxData 有几个成员来存组合框列表区的字串，选项数据，选项索引和编辑区的内容
Checkbox Radio button	WORD	BF_CHECKED，BF_UNCHECKED，BF_GRAYED
Scrollbar	TScrollBarData *	指向 TScrollBarData 实例的指针，TScrollBar 有三个 int 成员：LowValue 存范围的最小值，HighValue 存范围的最大值，Position 存当前 thumb 的位置

(一)列表框传输

由于列表框要传输几个信息(字串，某项目数据，选项索引)，传输缓冲区使用一个称为 TListBoxData 的类 TListBoxData 有几个成员来存取列表框的信息：

数据成员	类型	描述
ItemData	TDwordArray	包括项目的数据 DWORD
SelIndices	TIntArray	包含每个选择字串的索引
Strings	TStringArray	包括列表框中所有的字串

列表框还有成员函数来操纵列表框中的数据：

成员函数	描述
AddItemData	向 ItemData 中增加项目数据
AddString	向 Strings 中增加字串
AddStringItem	向 Strings 中增加字串，同时 ItemData 中增加项目数据
GetSelString	获取所给索引的被选字串
ResetSelections	从 Sellndices 数组中移去所有的选项
Select	选择所给索引的字串
SelectString	选择所给字串

(二)组合框传输

组合框需要传输几个信息(字串，项目数据、被选项目，被项目的索引)其传输缓冲区为 TComboBoxData。

数据成员	类型	描述
ItemDatas	TDwordArray	包含列表框中每项 DWORD 数据
Selection	char *	包含被选字串
Strings	TSttingArray	包含列表框的所有字串

TcomBox 有几个成员函数来操作组合框的信息

成员函数	描述
AddItem Data	向 ItemDatas 中增加数据
AddString	向 Strings 中增加字串
AddStringItem	向 Strings 中增加字串，同时向 ItemDatas 中增加索引

三、定义相应的窗口和对话框

窗口和对话必须构造参与传输的控制对象，其构造顺序必须和传输缓冲区中定义的成员一致。调用 SetTransferBuffer，传一个指向 transfer buffer 的指针参数来使传输起作用。

(一)使用传输对话框

由于对话框从资源中获取其控制的定义及本身的定义，所以应该用构造函数来构造控制。构造函数带一个资源 ID

```
Struet TTransferBuffer {
    char edit[30];
    TListBoxData * listBox;
    TScroll BarData * scrollBar;
};
...
TTransferDialog::TTransferDialog(TWindow * parent, int resId)
```

```
:TDialog(parent, resId),TWindow(parent)
{ new TEdit(this,ID_EDIT,30);
  new TListBox(this,ID_LISTBOX);
  new TScrollBar(this,ID_SCROLLBAR);
  SetTransferBuffer(& TTransfer Buffer);
}
```

这样构造的控制对象自动使传输成为可能(除开 button,groupbox,static control 外)。为了从传输机制中排除一个控制,调用 DisableTransfer 成员函数。

(二)用窗口传输

窗口中控制缺省时传输是不起作用的。为了将传输使能,调用 EnableTransfer 成员函数。

```
ListBox=new TListBox(this,ID_LISTBOX,20,20,340,100);
ListBox->EnableTransfer();
```

四、传输数据

绝大多数情况下,从窗口或向窗口传输数据都是自动的,但在任何时候,都可以显示传输数据。

(一)向窗口传输数据

当创建窗口对象时,向窗口传输数据都自动进行,构造函数调用 setupWindow 创建一个界面元素来代表窗口对象,它然后调用 TransferData 来从传输缓冲区中调入数据。窗口对象的 SetupWindow 调用子窗口的 SetupWindow。由于父窗口按照构造子窗口的顺序设置子窗口,所以传输缓冲区中的数据也要按照这种顺序出现。

(二)从对话框中传输数据

当模式对话框接收一个控制 ID=IDOK 的控制消息时,它自动从控制传送数据进缓冲区。通常,这个消息表示用户选择 OK 关闭了对话框,因此,对话框自动更新它的传输缓冲区。然后,再次执行对话框时,它从传输缓冲区向控制传输数据。

(三)从窗口传输数据

任何时候,都可以双向传输数据。例如,可能要从窗口或无模式对话框中传出控制的数据。有时,用户可能想恢复控制的初始状态,这些初始状态保存在传输缓冲区中。

在两种状态下使用 TransferData 成员函数。传送 IdSetData 从缓冲区向控制传输,传递 IdGetData 从控制向缓冲区传输。例如,调用 TransferData 成员函数(在 CloseWindows 中函数中)

```
void TMywindow::CloseWindow()
{ TransferData(tdGetData);
  TWindow::CloseWindow();
```

```
}
```

(四)支持用户自定义控制的传输

用户可能想修改特殊的控制传输数据的方式,或者在传输机制中定义新的控制。在两种情况下,只要为自定义控制对象写一个 Transfer 成员函数,下表解释了传输标志的含义。

传输标志参数	功能描述
tdGetData	将数据从控制拷入指针规定的位置。返回传送的字节数
tdSetData	将数据从缓冲区拷入控制返回传送的字节数
tdSizeData	返回将传输的字节数

2.6 界面对象

表示窗口,对话框、控制的 C++类的实例叫作界面对象。在屏幕上生成的窗口、对话框和控制叫作界面元素。本节将具体讨论界面对象的一般需求、功能以及与界面元素的关系。

实际上,界面对象包含在应用程序对象之中,它是一个 ObjectWindows 类,根据内存分配情况不同,界面对象存储在应用程序的堆或者栈中。而界面元素,实际上是 Windows 的一部分显示在屏幕上的实际窗口。

本节涉及到的信息适用于所有的界面对象,同时,本节将讨论不同界面对象之间的关系,界面对象响应 Windows 消息的机制。

2.6.1 为什么要用界面对象

开发 Windows 应用程序最大的困难是如何前后一致,有条不紊地控制诸界面元素。有时向窗口发送一个消息,有时又调用 Windows 的 API 函数,尽管操作类型很相近,但如果是针对不同的界面元素进行时,则约定也会不同。

如果利用 ObjectWindows 编程,则可使上述困难变得很容易解决。因为 ObjectWindows 提供了封装界面元素的对象。它使开发人员避免直接与 Windows 系统打交道,并为控制界面元素提供一个更加一致的界面。

一、界面对象做些什么

界面对象提供了许多成员函数,用来创建、初始化、管理以及撤销相关的界面元素,简化了程序员 Windows 编程的细节。

另外,界面对象还封装了与界面元素通讯所需要的数据,例如句柄,指向子窗口和父窗口的指针等。

界面对象与界面元素之间的关系同文件与 C++的流对象之间的关系十分相似。流对象仅代表一个存储在磁盘上文件,程序员通过流对象进行文件的读写等操作。对于 ObjectWindows,界面对象代表了 Windows 管理的界面元素。程序员与界面对象打交道,Windows 管理并维护界面元素。

二、通用界面对象:TWindow

ObjectWindows 所有界面对象都是从 TWindow 类派生出来的。它定义了所有的窗口、对话框和控制对象的共有属性。例如,TFrameWindow、TDialog、TControl 都是从 TWindow 派生出来的,并根据各自的需要,改进了 TWindow 的功能。

为所有界面对象的一个通用基类,TWindow 提供了一致的方法:

■ 维护界面对象与界面元素之间的关系,其中包括创建与析构界面对象与界面元素。
■ 处理界面对象与界面元素之间的父--子关系。
■ 注册新的 Windows 窗口类。

2.6.2 创建界面对象

创建一个界面对及相关的界面元素需要按如下两步进行:

1. 调用界面对象的构造函数创建界面对象并设置界面对象的属性。
2. 通知 Windows 生成具有新的界面元素的界面对象,同时创建界面元素。

 ■ 创造大多数的界面元素时,需要调用界面对象的 Create 成员函数。该函数间接调用 SetupWindow 函数,SetupWindow 通过创建一个界面元素,如子窗口,来初始化界面对象。
 ■ 创建一个有模式对话框,需要调用界面对象的 Execute 成员函数。

界面对象与界面元素之间关系是通过界面对象的数据成员 HWindow 来维护的。HWindow 是一个窗口句柄。

一、窗口句柄何时有效

通常在 Windows 环境下,一个新创建的界面元素通过自身的初始化来响应由 Windows 发送来的消息 WM_CREATE。ObjectWindows 的界面对象解释 WM_CREATE 消息然后调用 SetupWindow 函数执行用户定义的初始化工作。

如果界面对象的初始化工作需要界面元素的窗口句柄,则必须先调用基类的 SetupWindow 函数。在此之前,父窗口、子窗口均未创建,HWindow 无效,不能使用。可通过检查 HWindow 是否为 NULL 判断其是否有效,若为 NULL,则是无效的句柄,不能使用。不能把所有的初始化工作都集中在界面对象的构造函数内部完成,这似乎有点古怪,其实这是有道理的:如果一个界面元素已经创建,则它许多的特性是不能更改的。因此需要两个阶段的初始化工作,即在界面元素生成之前和界面元素创建之后。

在界面对象的构造函数中执行第一个阶段的初始化,在 SetupWindow 函数中执行第二阶段的初始化。为了便于理解,也可将 SetupWindow 看成是构造函数的第二部分。

二、使界面元素可见

创建界面对象及相应的界面元素并不意味马上可以在屏幕上看到它们。当 Windows 创造一个界面对象时,它首先检查界面的类型是否包含了 WS_VISIBLE,若是,则显示新创造建的界面元素,否则不显示。

TWindow 的构造函数设置 WS_VISIBLE 属性,因此大多数的界面对象在缺省情况下都是可见的。但是如果界面对象装入了一个资源,而且资源的类型定义包含了 WS_VISIBLE,则创建的界面对象可见;否则不可见。程序员可以通过界面对象的数据成员 Attr.Style 来设置 WS_VISIBLE 属性。

例如,应用 TDialog 装入一个没有设置 WS_VISIBLE 属性的对话框资源,在调用 Create 函数显示它之前,必须显示地设置 WS_VISIBLE 属性。

通过调用 IsWindowVisible 函数可获所创造的界面对象是否可见,若该函数返回值为 TRUE,则可见,若为 FALSE,则不可见。

界面元素创建成功后,调用成员函数 show 可以将其显示或隐藏。若为 show 提供一个"TRUE"参数,则执行显示功能,若提供一个"FALSE"参数,则执行隐藏功能。

三、界面对象特性

作为 ObjectWindows 对象,界面对象除了具有界面元素的属性外,还有一些其他的属性,可调用如下函数进行查询和修改:

- SetFlag　设置标志
- ClearFlog　清除标志
- IsFlagSet　检查标志是否已经设置。若该函数返回值为 TRUE,则指定标志已经设置,否则尚未设置。

可以应用的标志有:

- wfAlias　指示该对象是否有一个别名。
- wfautoCreate　指示是否自动生成该对象。
- wfFromResource　指示界面元素是否从资源文件中装入。
- wfShrinkToClient　指示框架窗口是否缩小以适应客户窗口的尺寸。
- wfMainWindow　指示该窗口是为主窗口。
- wfPredefinedClass　指示该窗口是否为预定义窗口类。
- wfTransfer　指示该窗口是否可以使用数据传输机制。

2.6.3 窗口特性

TWindow 提供了许多可用来改变界面元素资源与特性的函数。由于 TWindow 为众多的对象提供通用的功能,因此不会为操纵资源和特性提供具体的函数。较高层次的对象为此提供了十分具体的函数,但所有这些都是建立在 TWindow 提供的功能的基础之上,或是对 TWindow 功能的扩充。

- SetCaption　设定窗口标题为指定的字符串,该字符串以参数的形式传给 SetCaption 函数。
- GetWindowTextTitle　以字符串的形式返回当前窗口标题。
- SetCursor　按照指定的资源设置实例光标。
- 通过为 Attr·AccelTable　指定一资源 ID,可以为窗口安装一个加速器表。例如,假

设已有一个加速器表 MY_ACCELS，可按下述方法将资源赋给 Attr. AccelTable：

```
TMyWnd::TMyWnd(const char * title)
    { Init (0,title);
      Attr.AccelTable=MY_ACCELS;        // 赋键盘加速表
    }
```

关于以上各函数的详细内容，请参阅联机帮助。

2.6.4 撤销界面对象

撤销界面对象需要按如下两个步骤进行

- 撤销界元素。
- 删除界面对象。如果以后还会需要创建和显示某一界面元素，可以仅撤销界面元素，则不删除相应的界面对象。

一、撤销界面元素

界面对象的成员函数 Destroy 通过调用 API 函数 DestroyWindow 来撤销界面元素。当界面元素被撤销以后，相应的界面对象的数据成员 HWindow 被赋值零。因此，可以通过检查 HWindow 的值获知界面对象是否正与一个有效的界面元素相关联。

当用户关闭一个屏幕上的窗口时，会发生下述几件事：

- Windows 通知该窗口
- 该窗口通过调用本窗口及所有的子窗口的 CanClose 函数来确定该窗口是否能够关闭。
- 如果 CanClose 允许关闭该窗口，则界面元素，被撤销，相应的界面对象被删除。

二、删除界面对象

如果为了以后再次使用而撤销了界面元素，没有删除界面对象，那么在任务完成后，必须将界面对象删除。因为界面对象与常规的 C++对象没有什么区别，所以如果是应用 new 动态建立的对象，则可用 delete 将其删除。

下面的程序代码演示了如何撤销界面元素及删除界面对象：

```
TWindow * Window=new TWindow (0,"Sample Window");
//...
Window-> Destroy();
delete Window      // 撤销对象
```

2.6.5 父子界面元素

在 Windows 应用程序中，界面元素之间按父子关系组织在一起协调地工作，父窗口控制子窗口。Windows 记录这种链接关系。ObjectWindows 在相关的界面对象之间保存一组并行的链接关系。

子窗口是被另一个界面元素管理的界面元素。例如，某一窗口或对话框中的列表框。子

窗口只有在父窗口是可见的时候才能显示出来。前面提到的对话框是列表框的父窗口，同时它又是创建它的窗口的子窗口。

当移动或者关闭父窗口时，子窗口自动跟随父窗口移动和关闭。在应用程序中，最终的父窗口是应用程序的主窗口(例外的情况：可以创建没有父窗口的窗口对话框；所有的应用程序主窗口都是 Windows 桌面的子窗口)。

一、子窗口列表

当创建一个子窗口时，必须为其构造函数指定父窗口。子窗口利用数据成员 Parent 记录其父窗口。父窗口利用私有数据成员 childList 记录其子窗口。每一个父窗口都自动地保存着其子窗口的列表。

利用窗口包容类成员函数 FirstThat，ForEach 可以存取父窗口的所有子窗口。

二、创建子窗口

同所有的界面对象一样，创建子窗口对象也需要分两步走：构造界面对象和创建界面元素。在父窗口的构造函数中创建子窗口对象时，如果设置了子窗口对象的自动生成标志，则子窗口将随着父窗口一同创建。在缺省情况下，除了 TDialog 类之外，所有的 TWindow 派生类的 ObjectWindows 对象都具有自动生成功能，即 wfAutoCreate 标志被置位。

例如，基于 TWindow，创建一个带有三个按钮(子窗口对象)的窗口对象的构造函数为：

```
TTestWindow::TTestWindow(TWindow *parent,const char far *title)
{ Init(parent,title);
  Button1=new TButton(this,ID_BUTTON1,"Show",
                      190,270,65,20,FALSE);
  Button2=new TTestButton(this,ID_BUTTON2,"Hide",
                      275,270,65,20,FALSE);
  Button3=new TButton(this,ID_BUTTON3,"Transfer",
                      365,270,65,20,FALSE);
}
```

在上面的代码中，this 指针将子窗口与父窗口联系起来，子窗口的构造函数自动将它们自加到父窗口的子窗口列表当中。当 TTestWindow 的实例被创建以后，三个作为子窗口的按钮自动显示在父窗口内。

三、创建子界面元素

如果子窗口对象不在父窗口的构造函数中创建，则当父窗口创建成功并显示在屏幕上时，子窗口不会马上显示出来。但用户可调用 Create 函数自己创建。对于有模式对话框，需要调用 Execute 函数。

例如，欲创建一个带有两个按钮的窗口，这两个按钮随主窗口一同显示出来，一个标记"Show"，另一个标为"Hide"，当用户按下"Show"按钮时，显示第三个按钮，标记为"Transfer"；当用户按下"Hide"按钮时，撤销"Transfer"按钮。程序代码如下：

```
class TTestWindow:public TFrameWindow
```

```
{ TButton  * button1; * button2, * button3;
   public :
        TTestWindow(TWindow  * parent,const char far  * title);
   void EvButon1()
   { if (!button3->HWindow)
        button3->Create();
   }
   void EvButton2()
   { if (button3->HWindow)
        button3->Destroy();
   }
   void EvButton3()
   { MessageBeep(-1);
   }
   DECLARE_RESPONSE_TABLE(TTestWindow);
};
DEFINE_RESPONSE_TABLE1(TTestWindow,TFrameWindow)
   EV_COMMAND(ID_BUTTON1,EvButton1),
   EV_COMMAND(ID_BUTTON2,EvButton2),
   EV_COMMAND(ID_BUTTON3,EvButton3),
END_RESPONSE_TABLE;
TTestWindow:TTestWindow(TWindow  * parent,const char far  * title)
{ Init(parent,title);
   button1=new TButton(this,ID_BUTTON1,"Show",
                     10,10,75,25,FALSE);
   button2=new TTestButto(this,ID_BUTTON2,"Hide",
                       90,10,75,25,FALSE);
   button3=new TButton(this,ID_BUTTON3,"Transfer",
                     180,10,75,25,FALSE);
   button3->DisableAutoCreate();
}
```

在 TTestWindow 的构造函数中调用 DisableAutoCreate 函数是为阻止"Transfer"按钮随着窗口一同显示出来。函数 EuButton1和 EuButton2都是通过测试 Button3的数据成员 HWindow 来工作的。只有当"Transfer"按钮没有显示时,EvButton1函数才会将其显示出来,另外,只有显了"Transfer"按钮,EvButton2函数才会将其撤销。

四、撤销窗口

撤销一个父窗口的同时,也就撤销了它的所有子窗口。没有必要逐一去删除所有的子窗口界面对象。其道理同 CanClose 机别是一样的。父窗口的 CanClose 函数调用所有子窗口的 CanClose 函数,只有所有子窗口的 CanClose 函数都返回 TRUE,父窗口的 CanClose 才返回 TRUE。

当撤销一个对象的界面元素时,它自动为所有的子对象设置自动生成标志,而不管该标志以前的值如何。这样,再次生成父对象时,所有的子对象又被恢复到原来的位置上。基于此,可以撤销一个界面元素,然后在原来的地方再次创建它。

为了避免不必要的自动生成,必须显示地对所有的子对象关闭其自动生成标志。

五、自动生成

在创建父对象之前若设置了自动生成标志,则子对象将随着父对象的创建一同创建。这一点对于所有的子对象而言都是毫无意义的。

调用子对象的成员函数 DisableAutoCreate 可以将该子对象排除在自动生成一和一显示机制之外。调用 EnableAutoCreate 函数能把子对象到自动生成一和一显示机制当中。

除对话框类之外,所有 ObjectWindows 类的自生成标志都是使能的。

六、操作子窗口

TWindow 提供了两个包容类函数,ForEach 和 FirstThat 应用这两上函数即可以对所有子对象,也可以对一个特定的子对象进行操作。另外,TWindow 还提供了其他的用以确定子对象数目以及遍历子对象列表的函数。

1. 对所有的子对象操作:ForEach

有时需要对所有的子窗口对象进行一定的操作。这时需要用 ForEach 函数。它带有一个参数:一个指向函数的指针,指针所指向的函数即可以是某一对象的成员函数,也可以是一个单独的函数。无论是哪一种情况,这个函数必须带有两个参数,即 TWindow * 和 void * 。对每一个子对象,ForEach 都调用该函数。子对象以 TWindow * 的方式传递给该函数;void * 参数的缺省值为零。另外也可用它向用户自定义的函数传递其他类型的参数。

调用了用户定义的函数之后,处理子对象必须十分小心。尽管子对象按 TWindow * 的方式传递,实际上它是 TWindow 的一个后代。为了确保处理正确无误,应该用宏 DYNAMIC_CAST 把 TWindow * 转换为 TClass * ,TClass 代表子对象的类型。

例如,欲检查父窗口中的所有复选框(checkbox),代码如下:

```
void
CheckTheBox(TWindow *win,void *)
{ TCheckbox *cb=DYNAMIC_CAST(win,TCheckbox);
  if(cb)
  cd->Check();
}
void TMDIWindow::CheckAllBoxes()
{ ForEach(CheckTheBox);
}
```

如果欲转换成的类型是从基类虚拟派生出来的,则必须应用 DYNAMIC_CAST 进行转换。在本例中,TCheckbox 不是虚拟派生的。使用 DYNAMIC_CAST 进行转换是多余的。

如果不能进行类型转换,DYNAMIC_CAST 返回值为0,这一点很有用,因为并不是所

有对象都是 TCheckbox 类型。如果遇到了一个 TControlBar 对象,cb 的值为0,从而可以避免对其进行检查。

2. 寻找某一个特定的子对象

有时会遇到要处理某一个特定子窗口的问题。此方法有一个父窗口,在父窗口内有若干复选框,现在想找到第一个复选框,需要用 TWindow::FirstThat 函数来实现:

```
BOOL
IsThisBoxChecked(TWindow * cb,void * )
{ return cb?
  (cb->GetCheck==BF_CHECKED):
  FALSE;
}

TCheckBox *
TMDIFileWindow::GetFirstChecked()
{ return FirstThat(IsThisBoxChecked);
}
```

3. 管理子对象列表

除了包容类函数 ForEach 和 FirstThat 之外,TWindow 还提供了其他几个函数,用以定位和操纵某一个特定的子窗口。

- NumChildren,返回一个 unsigned 数,指出子窗口总数
- GetFirstChild 返回一个 TWindow *,指向子对象列表的第一项。
- GetLastChild 返回一个 TWindow *,指向于对象列表的最后一项。
- Next 返回一个 TWindow *,指向子对象列表中的下一项。
- Previous 返回一个 Window *,指向子对象列表中的前一项。

2.6.6 注册窗口类

无论何时调用 Create 或者 Execute 函数创建界面元素,界面对象首先检查是否已有同种类型的对象向 Windows 注册过。若有,则根据已注册的类创建界面元素。若没有,则先注册自己,然后创建界面元素。

为此,程序员在应用窗口类时,没有必要担心所用窗口类尚未注册之类的问题。

2.7 菜单对象

对大部分应用程序来说,在进行主窗口初始化时,需要为其定制一个菜单。在这一方面,与其他的应用开发工具相比,ObjectWindows(TMenu 类、TSystemMenu 类和 TPopupMenu 类以及 TMenuDescr 结构)为开发人员提供了非常简单的方法,可用于菜单的创建及应用。

本节讨论如下有关菜单对象方面的问题:

- 创建菜单对象。

- 修改菜单对象。
- 查询菜单对象。
- 应用系统菜单。
- 应用弹出式菜单。

2.7.1 创造菜单对象

TMenu 类提供了若干构造函数,利用这些函数,既可以根据菜单资源创建菜单,也可以根据已生成的窗口创建菜单。菜单创建完以后,可以执行菜单的增加、删除、修改等操作。下面列出了可用的构造函数。

表2.14 TMenu 类提供的创建菜单的构造函数

TMenu 构造函数	功能描述
TMenu()	创建一空的菜单
TMenu(HWND)	创建一个代表窗口当前菜单的菜单对象
TMenu(HMENU)	根据已装入内存的菜单创造一个菜单对象
TMenu(PCVOID *)	根据内存中的菜单模板创建一个菜单对象
TMenu(HINSTANCE,TResID)	根据给定的资源创建一个菜单对象

2.7.2 修改菜单对象

创建了菜单对象,可调用下表列出的 TMenu 的成员函数对其进行修改:

表2.15 修改菜单对象的 TMenu 类成员函数

TMenu 的成员函数	功能描述
增加菜单选项:	
AppendMenu(UNIT,UNIT, const char *)	在菜单选项的末尾增加一个选项
AppendMenu(UNIT,UNIT, const Tbitmap &)	在菜单选项的末尾增加一个位图选择项
InsertMenu(UNIT,UNIT,UNIT, const char *)	在指定选项的后面增加一个选项
InsertMenu(UNIT,UNIT,UNIT, const Bitmap!)	在指定选项的后面增加一位图选项
修改菜单选:	
ModifyMenu(UNIT,UNIT,UNIT, const char *)	修改指定的菜单选项
ModifyMenu(UNIT,UNIT,UNIT, const TBitmap)	将指定的选项改为位图
使能菜单选项:	
EnableMenuItem(UNIT,UNIT)	使指定的选项可选或不可选

(续)表2.15

TMenu 的成员函数	功能描述
删除和移动菜单选项:	
DeleteMenu(UNIT,UNIT)	删除一个弹出菜单或一个指定的菜单选项
RemoveMenu(UNIT,UNIT)	从菜单中而不是从内存中将指定的菜单选项移走
校验菜单选项:	
CheckMenuItem(UNIT,UNIT)	校验或不校验菜单选项
SetMenuItemBitmaps(UNIT,UNIT,const TBitmap * ,const TBitmap *)	根据指定的选项校验与否显示不同的位图
显示弹出菜单:	
TrackPopupMeun(UNIT,int,int,int,HWND,TRect *)	在给定窗口的指定位置弹出一菜单
TrackPopupMenu(UNIT,TPoint,int,HWND,TRECT *)	在给定窗口的指定位置弹出一菜单

改完菜单对象,应该调用窗口对象的 DrawMenuBar 函数根据所做的修改更新屏幕上显示的菜单选项。

2.7.3 查询菜单对象

TMenu 提供了一些可供查询菜单对象的函数。在修改菜单对象之前,经常需要调用这些函数。例如,在调用 SetMenuItemBitmaps 函数之前应用调用 GetMenuMarkDimensions 函数。

下表列出了有关查询方面的函数:

表2.16 用于查询菜单对象的 TMenu 成员函数

TMenu 成员函数	功能描述
查询整个菜单对象:	
操作符 UINT()和 HMENU()	返回菜单句柄
ISOK()	检查菜单是否有一个有效的句柄
GetMenuItemCount()	返回菜单选项个数
GetMenuCheckMarkDimensions(TSize &)	返回用于显示校验标志的位图尺寸
,本义菜单选项	
GetMenuItemID(int)	返回指定位置的菜单选项标识(ID)
GetMenuState(UNIT,UINT)	返回指定选项的状态标志
GetMenuString(UINT,char * ,int,UINT)	获取指定菜单选项的文本信息
GetSubMenu(int)	返回指定位置处的菜单的句柄

2.7.4 应用系统菜单

借助于 ObjectWindows 的 TSystemMenu 类,可以修改窗口的系统菜单。TsystemMenu 从 TMenu 派生出来,其构造函数需要两个参数,即一个窗口句柄和一个标志。如果标志的

值是 TRUE,当前系统菜单被删除,并产生一个表示原菜单的菜单对象,如果该标值为 FALSE,则创建一个代表当前系统菜单的菜单对象。

所有从 TMenu 类继承来的成员函数都可应用于系统菜单。

2.7.5 应用弹出菜单

利用 TpopupMenu 类可以创建一个弹出菜单,并把该弹出菜单加到一个已经存在的菜单结构中或者在一个窗口中直接使用。与 TSystemMenu 一样,TpopupMenu 从 TMenu 类派生出来,与 TMenu 类相比较,仅是构造函数不同。TPopupMenu 创建一个空的弹出菜单,然后可以为其添加所需要的选项。

利用 TrackPopupMenu 可以把已创建的一个弹出菜单作为自由的浮动菜单来显示。

2.7.6 为框架窗口加入菜单资源

在 ObjectWindows 1.0中,可以将一个菜单资源直接赋给框架窗口的成员 Attr.Menu,例如:

```
Attr.Menu=MENU_1;
```

在 ObjectWindows 2.0中,这样做是被禁止的,应该调用 AssignMenu 函数。AssignMenu 在 TFramWindow 中定义,可在 TFramWindow 及其派生类中应用,如 TMDIFrame、TMDIChild,TDecoratedFrame 和 TFloatFrame 等。

AssignMenu 函数需要唯一的一个 TResID 类型的参数,如果运行成功,返回值为 TRUE。由于 AssignMenu 被声明为 virtual,故可在 TFrameWindow 的派生类中对其重载。下面是应用 AssignMenu 函数对上述例子的具体实现:

```
AssignMenu(MENU_1);
```

BC 4.0 OWL 2.0和 Borland c++ 3.1 OWL 1.0相比,功能上改进和提高了很多。OWL 2.0将菜单作为一个对象处理,是充分利用了面向对象的程序设计思想,程序员只要在程序中创建一个菜单,操作菜单只要用该对象负责即可。

2.8 图形对象

ObjectWindows 2.0封装了 Windows GDI,使得应用 GDI 图形对象及函数更为方便。因为它简化了创建与应用 GDI 对象的方法。从简单的 GDI 对象,如画笔和刷子等,到复杂的 GDI 对象,字体和位图等,都是提供了简单、一致的图形编程模块。

2.8.1 GDI 类组织

有一些 objectWindows 类封装了 GDI 的功能。其中的大多数类都是从 TGdiObject 类派生出来的。该类为所有的 ObjectWindows GID 类提供了通用的功能支持。

TGdiObject 是 ObjectWindows GDI 对象的抽象基类。它提供了一个基类析构函数、一个转换操作符 HGDIOBJT、一个基类 GetObject 函数。另外,它还提供了对真 GDI 对象的独立控制(由 TGdiObject 在派生出来的对象叫作真 GDI 对象;其他的从 TGdiBase 类派生的对

象，如 TRegion、TIcon、和 TDib 等，称之为 GDI 对象)。

其他的 ObjectWindows GDI 类有：

- TDC 封装 objectWindows GDI 设备上下文的基类.既可以从 TDC 创建一个 TDC 对象，也可从 TDC 的派生类创建一个具有特殊功能的 TDC 对象。
- TPen 包含了 Windows 画笔的功能。可以根据一个已存在的画笔句柄、画笔对象、或者一个逻辑结构(LOGPEN)创建一个画笔对象。
- TBrush 包含了 Windows 刷子的功能。可以构造一个自定义刷子、创建一个实心刷子、网格刷子，或者使用一个已存在的刷子句柄、刷子对象、逻辑刷子结构(LOGBRUSH)。
- TFont 借助于 TFont 可以有效地利用 Windows 提供的各种字体。可以构造自定义的字体，或者根据已存在的字体句柄、字段对象、逻辑字体结构(LOGFONT)创建字体对象。
- TPalette 封装了一个 GDI 调色板。即可以构造一个新的调色板，或者应用由不同颜色表类型组成的调色板，DIB 对象使用该类型的调色板。
- TBitmap 封装了 Windows 的位图。可以利用多种资源创建位图，这些资源包括文件、位图句柄、应用资源等。
- TRegion 定义了窗口内的一块区域。可以利用各种形状来定义一个区域，如矩形、椭圆、多边形。TRegion 是一个伪 GDI 对象，因为它不是由 TGdiObject 派生出来的。
- TIcon 封装 Windows 的图形。可以利用资源或一些明确的信息构造一个图标。TIcon 是一个伪 GDI 对象。
- TCursor 封装了 Windows 的光标。可以利用资源或确切的信息构造一个图标。
- TDib 封装了设备无关位图类。设备无关位图(DIB)没有 Windows 句柄，它只是一个包含格式信息、调色板、位图数据的结构。该类为处理 DIB 提供了一个非常便利的方法。DIB 可以存储在.BMP 文件中，位图资源中、或者存到剪贴板上。TDib 是伪 GDI 对象。

2.8.2 对封装的 GDI 函数的修改

ObjectWindows GDI 类中的许多函数与常规的 Windows GDI 函数非常一致，如具有相同的名字或一致的函数声明。由于 ObjectWindows 需要重现许多 Windows 对象的功能，没有必要放弃已经很成熟的技术。因此有意将 ObjectWindows GDI 类中的函数与 Windows GDI 函数取相同的名字，一致的函数声明。

与标准的 Windows 函数相比，ObjectWindows 类中的函数做了适当的改进。这些改进包括分解打包的返回信息，应用 ObjectWindows 对象取代 Windows 定义的结构。

上述改进都是有一定理由的；一个函数可以改变并不意味着有必要改变它。如果发现一个 ObjectWindows 函数与 Windows API 函数名字相同，而看起来又有区别，那么很可能是出于下面的原因：

- Windows API 函数需要一个对象句柄做参数，而 ObjectWindows 类中的函数常常把它忽略掉。因为 TGdiObject 基对象保存了每个对象的句柄，当调用 Windows API 函

数时，保存的对象句柄将会传递给所调用的函数。例如，把一个对象选入设备上下文，通常要调用 Windows API 函数 SelectObject，如下所示：

```
void SelectPen(HDC & hdc,HPEN(hpen)
    {
        HPEN hpenold;
        hpenold=SelectObject(hdc,hpen);
        // Do something with the new pen
        ⋮
        // now select the old pen again
        SelectObject (hdc,hpenold);
    }
```

在 ObjectWindows 中，相对应的函数被封装在 TDC 中，TDC 由 TGdiObject 派生而来，下面的例子示范了在 TDC 的派生类对象的成员函数中如何调用 SelectObject 函数。同上面的例子进行比较，注意两次调用 SelectObject 的区别：

```
void SelectPen(TDC & dc,TPEN & pen)
    {
        dc.SelectObject(pen);
        // Do something with the new pen
        ⋮
        // now select the old pen
        dc.RestorePen();
    }
```

- ObjectWindows GDI 函数通常用 ObjectWindows 的数据类型取代 Windows 的数据类型：
- Windows API 函数应用两个单独的参数定义 x 和 y 坐标，而 ObjectWindows GDI 函数使用 TPoint 对象。
- Windows API 函数使用 RECT 结构定义一个矩形区域；而 ObjectWindows GDI 函数则使用 TRect 对象。
- Windows API 函数应用 RGN 结构定义一个区域；而 ObjectWindows GDI 函数则使用 TRegion 对象。
- 当向 Windows API 函数传递一个对象时，若该对象不具有预定义的 Windows 结构，则要使用 HLOCAL 或 HGLOBAL 参数。而 ObjectWindows GDI 函数则使用对 ObjectWindows 对象的引用。
- 一些 Windows 函数返回一个对数据进行过编码的 DWORD，只有对该 DWORD 进行解码，才能获取所包含的数据，而 ObjectWindows 函数以一个对适当对象的引用作为参数，该函数把需要返回的数据放入到该对象中，从而避免了程序员对数据进行解码。ObjectWindows 函数通常返回一个 BOOL 值，该 BOOL 值表示函数调用是否成功。

例如，Winbows 函数 SetViewPortOrg 返回一个 DWORD 值，在该 DWORD 值内包含了

视口原点坐标。相对应的 ObjectWindows 函数 SetViewPortOrg 的参数是两个对 TPoint 对象的引用。第一个 TPoint 对象包含欲设定的视口原点坐标，第二个 TPoint 返回原视口原点坐标。

2.8.3 应用设备上下文

当应用 Windows GDI 时，需要通过设备上下文与所有的设备打交道，从窗口到打印机到绘图仪，无一例外。设备上下文是 GDI 保存的一个数据结构。它包含了相应设备的一些必要信息。如缺省的前景颜色、背景颜色、字体、调色板等。ObjectWindows 2.0把设备上下文信息封装在几个设备上下文类中，所有的设备上下文类都是 TDC 的派生类。

TDC 封装了可能会用到的几乎所有的设备上下功能。其他的设备上下文类都是 TDC 的派生类，或者是 TDC 派生类的派生类。这些派生类进一步专门化了 TDC 类的设备上下文功能。下面简要介绍一下每一个设备上下文类。

- TDC 是 ObjectWindows 设备上下文类的基类。即可应用 TDC 类，也可应用 TDC 的派生类。TDC 的派生类可以提供更为具体的专门化的功能。
- TWindowsDC 提供了存取窗口整个区域的方法。它是一些设备上下文类的基类，这种类的特点是：用完后必须释放设备上下文句柄。
- TScreenDC 提供了存取屏幕位图的方法，这样做时窗口句柄为零，意味着要处理的是整个屏幕。
- TDesktopDC 提供了存取桌面窗口客户区的方法。
- TClientDC 提供了存取整个窗口客户区的方法。
- TPointDC 在 WM_PAINT 消息响应函数中包含了对 BeginPaint 和 EndPaint 的调用。
- TMetaFileDC 为设备上下文提供了一个已装载的元文件。
- TCreatedDC 可以为指定的设备创建一个设备上下文。
- TIC 可为指定的设备创建一个信息上下文。
- TMemoryDC 提供了存取内存设备上下文的途径。
- TDibDC 借助于 TDibDC，可以利用驱动程序 DIB. DRV 提供的功能存取 DIB(设备无关位图)。
- TPrintDC 提供了存取打印机设备上下文的途径。

一、TDC 类

尽管 TDC 的派生类提供了附加的成功，但 TDC 类提供了其中的大部分功能，下面介绍其基本功能。

由于 TDC 中包含了众多的函数，这里不可能详细地介绍每一个函数，而仅介绍其功能及适用范围、但对于 ObjectWindows 中的一些特殊函数，以及与 Windows API 比较近似的函数在 TDC 类中进行了详细的讨论。关于其他的函数，可参阅《ObjectWindows Reference Guide》，特别地，TDC 中的大部函数与 Windows WPI 函数比较接近，因此在本节仅做简要介绍。

1. 构造与撤销 TDC

TDC 类提供了一个公有构造函数与析构函数。构造函数带有一个参数 HDC,即一个指向设备上下文的句柄。这意味着在创建一个 TDC 对象时,必须已经有了一个设备上下文。一般来讲,并不直接创建一个 TDC 对象。尽管可以这样做。在以参数的形式或指向 TDC 对象的指针的形式传递设备上下文时,通常使用指向某一设备上下文的 TDC 对象,该设备上下文包含在 TDC 类或 TDC 的派生类中。

~TDC 撤销选入设备上下文中的对象,恢复所有的缺省对象。

TDC 另外还提供了两个供派生类使用的受保护的构造函数、第一个是缺省的构造函数,派生类没有必要显示地调用它。第二个构造函数带两个参数:一个 HDC 和一个 TAutoDelete 标志。TAutoDelete 是枚举型(enum)数据,其值为 NoAutoDelete 或 AutoDelete。TAutoDelete 参数用来初始化 ShouldDelete 成员,ShouldDelete 由 TGdiObject 继承而来,(TDC 的公有的构造函数将 TAutoDelete 初始化为 NoAutoDelete)。

2. 设备上下文操作符

TDC 提供了一个转换操作符:HDC,该操作符通常是陷式使用的。可以返回一个 TDC 或 TDC 派生类对象的设备上下文句柄。当应用 TDC 对象时,在通常需要使用 HDC 的地方,如函数调用等,编译器通过适当处理将指定的对象转变为需要的类型,在这种情况,就用到了 HDC,而且是隐式使用的。

例如,在内存中创建一个某一设备相兼容的设备上下文,该设备与一个 TDC 对象相联系:可以利用已经存在的 TDC 对象,调用 Windows API 函数 CreateCompatibleDC 创建之:

```
HDC GetCompatDC(TDC &dc ,TWindow * Window)
{
    HDC compatDC;
    if(!(compatDC=CreateCompatibleDC (dc)))
    {
        Window·MessageBox("Coundnt Create compatible device
    context!","Failure",MB_OK|MB_ICONEXCLAMATION);
        return NULL;
    }
    else return compatDC;
}
```

在上面的程序代码中,CreateCompatibleDC 只需一个参数,一个 HDC,因此在调用它时,参数 dc 被转换为一个 HDC,这实际是通过隐式调用 HDC 来实现的。

3. 设备上下文函数

这些函数与同名的 Windows API 函数一样,用于存取设备上下文本身的信息。

利用 SaveDC 和 RestoreDC 可以存储和恢复设备上下文。下面的示范代码演示了如何正确使用这两个函数。注意在下面代码中,没有为 RestoreDC 指定一个 int 参数,而是使用了缺省值。

```
void TMyDC::SomeFunc(TDC&dc,int x1,inty1,int x2,int y2)
{
    dc.SaveDC();
    dc.SetMapMode(MM_LOENGISH);
    ⋮
    dc.Rectangle(x1,-y1,x2,-y2);
    dc.Restore DC();
}
```

另外,利用函数 ResetDC 可以设置设备上下文信息为 DEVMODE 结构所包含的值。ResetDC 带一个参数,即一个对 DEVMODE 结构的引用。

利用 GetDeviceCaps 可以返回给定设备的设备上下文信息。该函数带一个参数,即一个用来索引设备上下文信息类型的整型数(int)。该参数值与 Window API 函数 GetDeviceCaps 的参数值相同。

利用 GetDCOrg 函数,可以定位当前的设备上下文在显示设备的绝对物理坐标系中的逻辑坐标值。该函数以一个对 TPoint 对象的引用作为参数。返回一个 BOOL 值。返回的 BOOL 值指明函数调用是否成功。TPoint 对象包含了返回的设备上下文的逻辑坐标。

4. 选择与恢复 GDI 对象

应用 SelectObject 函数可以把一个 GDI 对象选入设备上下文。共有四个不同的 SelectObject 函数,它们均无返回值,不同之处为:彼此需要不同的参数。应该使用哪一个函数,这取于欲选入设备上下文的 GDI 对象类型,四个 SelectObject 函数。

```
SelectObject(const TBrush&brush);
SelectObject(const TPen&pen);
SelecObject(const TFont& font);
SelectObject(const TPalette& palette,BOOL force RG=FALSE);
```

另外,TMemoryDC 可用来选择位图。

能够选入设备上下文的图形对象通常以逻辑对象的形式存在,它包含了创建实际的对象所需要的信息。图形对象是通过一个 Windows 句柄与逻辑对象相关联的,当图形对象被选入设备上下文时,在设备上下文内部会产生一个工具(根据在逻辑笔中保存的属性创建)。

应用 SelectStockObject 函数可以选择一个库存对象。该函数与 Windows API 的 SelectStockObject 函数一样,都需要一个 int 参数,调用该函数取代了调用如下两个函数:调用 GetStockObject 得到一个库存对象,然后调用 SelectObject 将该对象选入设备上下文。

TDC 提供了如下几个函数恢复设备上下文的初始对象:RestoreBrush,RestorePen、RestoreFont、RestorPalette,以及 RestoreTextBrush,该函数仅适用于32位应用程序。RestoreObject 函数调用前面四个函数恢复设备上下文的初始对象(若是32位应用程序,则上述五个函数均可用),上述几个函数均无返回值,也不需要为之提供参数。

5. 绘图工具函数

GetBrushOrg 函数需要一个参数,即一个对 TPoint 对象的引用。它把刷子的原点坐标放到 TPoint 对象中。如果调用成功,返回值 TRUE。

SetBrushOrg 函数需要两个参数，一个对 TPoint 对象的引用和一个指向 TPoint 对象的指针。该函数根据第一个 TPoint 对象的 x，y 值设置刷子的原点坐标，如果没有提供第二个参数，其缺省值为0；如果提供了第二个参数，TDC：setBrushOrg 把刷子原来的坐标原点存入该对象中。该函数返回一个 BOOL 值，根据返回的 BOOL 值可以判定函数调用是否成功。

6. 颜色(color)与调色板(palette)函数

TDC 提供了下面的函数，可用来使用设备上下文中的颜色及调色板。

GetNearestColor	RealizePalette
GetSystemPaletteEntries	SetSystemPalettes
GetSystenPaletteUs	UpdateColors

7. 绘图属性函数

利用下面的函数可以设置设备上下文的绘图模式。除了忽略了 HDC 参数之外，所有这些函数与 Windows API 函数都具有相同的名字。

GetBkColor	SetBkColor
GetBkMode	SetBkMode
GetPolyFillMode	SetPoly Fill Mode
GetPOR2	SetROP2
GetStretchBltMode	SetStretch BltMode
GetText Color	SetTextColor

另外，还有一个仅适用于32位应用程序的函数 SetMiterLimit。

8. 视口与窗口的图形函数

借助于下面的函数可以设置窗口和视口的图形模式：

GetMapMode	GetViewportExt
GetViewportOrg	ScaleViewportExt
ScalWindowExt	GetWindowExt
SetMapMode	GetWindowOrg
SetViewportExt	SetViewportOrg
SetWindowExt	offsetViewportOrg
SetWindowOrg	

下面两个函数仅适用于32位应用程序：

ModifWorld Transform	SetWorldTransform

9. 坐标函数

坐标函数将逻辑坐标值转为物理坐标值：

DPtoLP	LPtoDP

10. 剪切与更新矩形和区域的函数

利用下面的函数可以在设备上下文中的裁剪区域中设置和返回简单或复杂的区域：

ExcludeClipRect	offsetClipRgn
ExcludeUpdateRgn	PtVisible
GetBoundRet	RectVisible
GetClipBox	SelectClipRgn
GetClipRgn	SetBoundsRect
IntersetClipRect	

11. 元文件函数

下面是用于存取元文件用的元文件函数：

EnumMetaFile	PlayMetaFileRecord
PlayMetaFile	

12. 定位当前点函数

应用下面的函数可在设备上下文中移动当前位置，三个 MoveTo 函数为：

- MoveTo(int x,int y)把画笔移到点(x,y)处。
- MoveTo(TPoint &point)把画笔移到(point. x,point. y)处。
- MoveTo(TPoint &Point,TPoint &oldPoint)把画笔移到(point. x,point. y)处，并把移动前画笔的位置存到 oldPoint 对象中。

GetCurrentPosition 带一个参数，即一个对 TPoint 对象的引用，把当前总坐标存入 TPoint 对象中。如果调用成功，返回值 TRUE。

13. 字体函数

应用 TDC 的字体函数可存取、使用各种字体：

EnumFont Families	GetCharWidth
EnumFonts	GetFontData
GetAspectRatioFilter	SetMapperFlags
GetCharABCWidths	

14. 路径函数

路径函数仅适用于32位应用程序，下面的函数省略了 HDC 参数，除此之外，与对应的 Win32函数完全一致：

■ Icons	■ Shaps
■ Rectangles	■ Bitmaps
■ Regions	■ Text

几乎每一个函数都提供了不同的版本：一个与相应的 Windows API 函数功能一致(但忽略了 HDC 参数)，另外的版本使用 TPoint、TRect、Tregion 等 Windows 数据，从而使函数调用简明、易于理解。关于这些函数的细节请参阅《ObjectWindows Reference Guide》。

■ 当前位置(current position)

GetCurrentPositionMoveTo

■ 图标(Icons)

DrawIcon

■ 矩形(Rectangles)

DrawFocusRect	FillRect
FrameRect	TextRect
InvertRect	

■ 区域(Regions)

FillRgn	InvertRgn
FrameRgn	PaintRgn

■ 外形(Shapes)

Arc	Polaygon
Chord	Polyline
Ellipse	Polypolygon
LineDDA	Rectangle
LineTo	RoundRect
Pie	

■ 位图(Bitmaps)

BitBlt	ScrollDC
ExtFloodFill	SetDIBits
FloodFill	SetDIBitsToDevice
GetDIBits	SetPixel
PatBlt	StretchDIBits

■ Text(文本)

DrawText	TabbedTextOut
ExtTextOut	TextOut
GrayString	

下面的函数适用于32位应用程序:

■ Shapes

Aangle Arc	PolyDraw
PolyBezier	PolylineTo
PolyBezierTo	

■ Bitmaps and blitting

MaskBlt	PlgBlt

二、设备上下文对象的数据及函数

这里讨论的数据成员与函数用于管理设备上下文本身,它们都是受保护的(Protected),

只能由 TDC 的派生类的对象使用。

■ ShouldDelete 指明当激活设备上下文对象的析构函数时，是否应删除设备上下文句柄。
■ Handle 保存设备上下文句柄。
■ OrgBrush、OrgPen，OrgFont 和 OrgPaltte 是设备上下文创建时的句柄，OrgTextBrush 仅适用于32位应用程序。
■ CheckValid 在设备上下文无效的情况下产生一个特例。
■ Init 在设备上下文对象创建时设置 OrgBrush，OrgPen，OrgFont 和 OrgPalette。如果在创建一个 TDC 派生类对象时，没有显示地调用 TDC 构造函数，则应当在派生类的构造函数中首先调用 TDC::Init。
■ GetHDC 利用 Hardle 返回一个 HDC。
■ GetAttributeHDC 功能同 GetHDC。如果欲创建一个具有多个设备上下文的对象，应该重载该函数以便之返回正确的值。
■ OWLFastWindowFrame 在创建窗口边界时经常用到。该函数调用尚未公开的 Windows API 函数 FastWindowFrame 或者 PatBlt。

2.8.4 TPen 类

TPen 封装了一个逻辑笔，包含了画笔的颜色、宽度、风格度。

一、构造画笔

可以直接指定画笔的颜色、宽度、风格来构造一个画笔；或者通过指定一个对 TPen 对象的引用或者指向一个 LOGPE 结构的指针来间接地构造画笔对象。用直接的方法创建一画笔对象具有指定的画笔属性。下面是一个示范：

```
TPen (TColor color,int width=1,int style=PS_SOLID);
```

style 参数的值可以是：PS_SOLID、PS_DASH、PS_DOT、PS_DASHDOT、PS_DASHDOTDOT、PS_NULL 或者 PS_INSIDEFRAME。有关这些值的详细讨论请参阅《Object Reference Guide》。

间接地创建画笔对象时，为新画笔拷贝已存在画笔的属性，下面是间接创建画笔的例子：

```
TPen(const LOGPEN far * logPen);
TPen(const TPen&);
```

另外，借助于已有的 HPEN 句柄，也可创建一个新的画笔对象。

```
TPen (HPEN handle,TAutoDelete dutoDelete=NOAutoDelete);
```

另外有两个仅适用于32位应用程序的构造函数，借助于这两个构造函数可以创建美术笔或几何笔：

```
TPen(DWORD penStyle,DWORD width,
     const TBrush& brush,DWORD styleCount,
```

```
        LPDWORD style);
    TPen(DWORD penStyle,DWORD width,
        const LOGBRUSH& logBrush,DWORD styleCount,
        LPDWORD style);
```

其中：

- PenStyle 是笔的 type、style、end cap 和 join 的组合：
  - Type 可以是 PS_GEOMETRIC 或者 PS_COSMETIC。
  - Style 可以为下列其中之一：

    PS_ALTERNATE　　PS_INSIDEFRAME
    PS_DASH　　PS_NULL
    PS_DASHDOT　　PS_SOLID
    PS_DASHDOTDOT　　PS_USERSTYLE
    PS_DOT

- End cap 仅用于几何笔，可为下列值其中之一：

    PS_ENDCAP_FLAT　　PS_ENDCAP_SQUARE
    PS_ENDCAP_ROUND

- Join 仅用于几何笔，合法值为：

    PS_JOIN_BEVEL　　PS_JOIN_ROUND
    PS_JOIN+MITER

- width 是笔的宽度
- brusth 或 logBrush 是对已存在的 TBrush 或 LOGBRUSH 对象的引用。
- styleCount 是 style 为组的尺寸(以 DWORD 为单位计算)；styleCount 对应于 PS_USERSTYLE 的值为0。
- style 是一个指向 DWORD 型数组的指针，该数组用来确定笔的模式。如果笔的风格为 PS_USERSTYLE，style 必须为 NULL。

二、存取画笔

利用 HPEN 句柄或 LOGPEN 结构对象存取画笔。把 TPen 对象作为参数，利用操作符 HPEN，可得到 TPen 对象的 HPEN 句柄。HPEN 操作符总是被隐式调用的：

```
HPEN GetHPen(TPen& pen)
{
    return pen;
}
```

上面代码自动利用转换操作符 HPEN 把 TPen 对象转变为正确的类型。

为了把一个 TPen 对象转换为一个 LOGPEN 结构，需要使用 GetObject 函数：

```
BOOL GetLogPen(LOGPEN far & logPen)
{
```

```
    TPen pen(TColor::LtMagenta,10);
    return pen.GetObject(logPen);
}
```

下列例子演示，如何利用具有 TDC 参数的笔画线：

```
void TPenDemo::DrawLine(TDC & dc,const TPoint & point,TColor & color)
{
    TPen BrushPen(color,PenSize);
    dc.SelectObject(BrushPen);
    dc.LineTo(point);
}
```

2.8.5 TBrush 类

TBrush 类封装了一个逻辑刷子，它包含了刷子的颜色（封装在一个 TColor 对象中）、宽度、风格、模式、位图等。

一、构造刷子

与构造画笔类似，可以通过指定颜色、宽度及风格采用直接的方法构造一个刷子，或者利用对 TBrush 对象的引用或指向 LOGBRUSH 结构对象的指针间接的构造一个刷子。采用直接的方法构造的刷子具有指定的属性。下面列出的是可用于直接创建刷子的构造函数：

```
TBrush(TColor solor);
TBrush(TColor color,int style);
TBrush(const TBitmap& pattern);
TBrush(const TDib& pattern);
```

第一个构造函数创建一个具有指定颜色的实心刷子。

第二个构造函数创建一个具有指颜色与风格的网格刷子。风格可以是：

```
HS_BDIAGONAL        HS_FDIAGONAL
HS_CROSS            HS_HORIZONTAL
HS_DIAG CROSS       HS_VERTICAL
```

第三、四个构造函数利用指定的位图创建一个位图刷子。刷子的宽度依赖于位图的尺寸。

用间接的方法创建一个新刷子，它拷贝现有刷子的属性。下面是两个构造函数：

```
TBrush(const LOGBRUSH far * logBrush);
TBrush (const TBrush& src);
```

另外，可利用现有的 HBRUSH 句柄创建一个新的 TBrush 对象：

```
TBrush (HBRUSH handle,TAutoDelete autoDelete=NoAutoDelete);
```

二、存取刷子

存取刷子时需要用到 HBRUSH 句柄或 LOGBRUSH 结构对象。把 TBrush 对象作为参

数，利用 HBRUSH 操作符，可以得到 TBrush 对象的 HBRUSH 句柄。HBRUSH 操作符总是被隐式调用的：

```
HBRUSH GetHBrush(TBrush& brush)
{
    return brush;
}
```

上述代码自动调用 HBRUSH 转换操作符将 TBrush 对象转变为正确的类型。

应用 GetOBject 函数可把一个 TBrush 对象转变为一个 LOGBRUSH 结构：

```
BOOL GetLogBrush (LOGBRUSH far& logBrush)
{
    TBRUSH brush (TColor::LtCyan,HS_DIAGCROSS);
    return brush.GetObject(logBrush);
}
```

应用 UnrealizeObject 函数可重置刷子的原点。如果成功，返回一个非零值。

下面的代码演示了如何用刷子在窗口中用刷子刷一个矩形：

```
void TMyWindow::PaintRect(TDC &dc,TPoint & P,TSize & size)
{
    TBrush brush(TColor (5,5,5));
    dc.SelectObject(brush);
    dc.Rectangle(p,size);
    dc.RestorBrush();
}
```

2.8.6　TFont 类

利用 TFont 类很容易在应用程序中创建和使用 Windows 字体。TFont 类封装了逻辑字体的所有属性。

一、构造 TFont 对象

可以通过指定字体的所有属性直接创建一个字体对象；或者利用对现有 TFont 对象的引用、或一个指向 LOGFONT 结构对象的指针来间接地创建一个 TFont 对象。用直接方法创建的字体对象具有规定的属性。下面是直接创建 TFont 对象的构造函数：

```
TFont(const char far * facename=0,
        int height=0,int width=0,int escapement=0,
        int orientation=0,int weight=FW_NORMAL,
        BYTE   pitchAndFamily=DEFAULT_PITCH|FF_DONTCARE,
        BYTE   italic=FALSE,BYTE underline=FALSE,
        BYTE   strikeout=FALSE,
        BYTE   charSet=1,
        BYTE   outputPrecision=OUT_DEFAULT_PRECIS,
```

```
        BYTE   cliPrecision=CLIP_DEFAULT_PRECIS,
        BYTE   quality=DEFAULT_QUALITY);
TFont (int height,int width,int escapement=0,
        int orientation=0,
        int height=FW_NORMAL,
        BYTE   italic=FALSE,BYTE undline=FALSE,
        BYTE   strikeout=FALSE,
        BYTE   charSet=1,
        BYTE   outputPrecision=OUT_DEFAULT_PRECIS,
        BYTE   clipPrecision=CLIP_DEFAULT_PRECIS,
        BYTE   quality=DEFAULT_QULITY,
        BYTE   pitchAndFamily=DEFAULT_PITCHIFF_DONTCARE,
        const char far * facename=0);
```

利用第一个构造函数,很方便规定一些常用的字体属性,而使其他的属性取其缺省值。第二个构造函数与 Windows API 函数 CreatFont 的参数顺序完全一致。因此可以剪贴已有的 Windows 代码。

间接创建字体对象时,拷贝现有字体对象的属性为新字体所有。下面是间接创建字体对象的构造函数:

```
TFont(const LOGFONT far * logFont);
TFont(const TFont&);
```

另外,利用下面的构造函数从已有的 HFONT 句柄创建一个字体对象:

```
TFont(HFONT handle,TAutoDelete dutoDelete=NoAutoDelete);
```

二、存取字体

应用 TFont 类需要借助于 HFONT 句柄或 LOGFONT 结构。利用 HFONT 操作符,把现有的 TFONT 对象作为参数,可得到该对象的 HFONT 句柄。与前面介绍的 GDI 类一样,HFONT 操作符总是被隐式调用的:

```
HFONT GetHFont (TFont& font)
{
    return font;
}
```

在上面的代码中,隐式地调用 HFONT 操作符将 TFont 对象转换为正确的类型——字体句柄。

利用 GetObject 函数可将一个 TFont 对象转化为一个 LOGFONT 结构,如下所示:

```
BOOL GetLogFont(LOGFONT far& logFont)
{
    TFont font("Times Roman",20,8);
    return font.GetObject (log Font);
```

```
}
```

2.8.7　TPalette 类

TPalette 类封装了位图及 DIBs(设备无关位图)所用的 Windows 的调色板。利用 TPalette,可以调整颜色表,匹配某一颜色、把调色板移入剪贴板等。

一、构造调色板

构造 TPalette 对象有两种方法,一种是直接的方法,即将一存有颜色值的数组传递给 TPaletle 的构造函数;另一种是间接的方法,这时需要指定一个对 TPaletle 对象的引用、或一个指向 LOGPALETTE 对象的指针、或者一个指向位图文件头的指针。用直接的方法创建的调色板具有指定的属性。下面是个直接构造 TPalette 对象的构造函数:

```
TPalette(const PALETTEENTRY far * entries,int count);
```

entries 是 PALETTEENTRY 对象数组,每一个 PALETTEENTRY 对象包含了某一颜色的红(R)绿(G)、蓝(B)分量,以及一个标志量。count 表示 entries 数组中对象的个数。

用间接方法创建的对象,它拷贝现有对象的属性,下面可用的构造函数:

```
TPalette(const TClipboard& );
TPalette(const TPalette& palette);
TPalette(const LOGPALETTE far * logPalette);
TPalette(const BITMAPINFO far * info,UNIT flags=0);
TPalette(const BITMAPCOREINFO far * core,UNIT flags=0);
TPalette(const TDib& dib,UNIT flags=0);
```

上面每一个构造函数为新创建的对象拷贝指定的颜色值。传递给上述构造函数的对象不一定是调色板对象,它们可以是应用调色板的对象。在这和中情况下,TPalette 构造函数能够将调色板从传递的对象中提取出来,并拷贝给新创建的对象。

另外,利用现有的 HPALETTE 句柄也可创建一个新的 TPalette 对象:

```
TPalette (HPALETTE handle,TAutoDelete AUtoDelete=NoAutoDelete);
```

二、存取 TPalette

存取 TPalette 需要借助于 HPALETTE 或 LOGPALETTE 结构。利用 HPALETTE 操作符,将 TPalette 对象作为参数,可以得到该对象的 HPALETTE 句柄,此时,HPALETTE 是被隐式调用的:

```
HPALETTE GetHPalette(TPalette & palette)
{
    return palette;
}
```

上面的代码隐式调用 HPALETTE 转换操作符将 TPalette 对象转换为 HPALETTE 类型。

TPaletee 的 GetObject 函数与 Windows API 函数 GetObject 的工作方式一致。

Tpalette 封装了一些标准的 API 调用：

- 利用 GetNearestPaletteIndex 函数可以在调色板中找到与某一颜色最匹配的值。该函数带一个参数（一个 TColor 对象），返回调色板中匹配值的索引。
- GetNumEntries 不带任何参数，返回调色板颜色表中的颜色数目。
- GetPaletteEntries 函数用于返回调色板中颜色表的若干项数。除了省略了 HPALETTE 参数之外，GetPlaetteEntries 函数与 Windows API 的 GetPaletteEntries 的调用方式一致。
- SetPaletteEntries 函数用于设置调色板颜色表中的若干项。除了省略了 HPALETTE 参数之外，调用 SetPaletteEntries 的方法与调用 Window API 的 SetPaletteEntries 方法一样。
- GetPaletteEntry 和 SetPaletteEntry 函数与 GetPaletteEntries 和 SetPaletteEntries 的功能十分相似。只不过是前者每次操作的是颜色表中的一个表项。这两个函数均带两个参数，一个是调色板表项的索引，一个是对 PALETTEENTRY 对象的索引。GetPaletteEntry 把指定的表项包含的颜色放入 PALETTEENTRY 对象中。SetPaletteEntry 利用 PALETTEENTRY 对象设置指定的颜色表项。
- ResizePalette 函数用于改变调色板的尺寸。它带一个 UINT 型参数。定义改变后的调色板尺寸。它与 Windows API 中的 ResizePalette 函数的功能完全一致。
- AnimatePalette 函数用于替换调色板的颜色表。它带三个参数，即两个 UINT 参数和一个指向 PALETTEENTRY 对象数组的指针。第一个 UINT 型参数定义欲替的第一个表项，第二个 UINT 型参数定义欲替换的项目数。用于替换的值保存在 PALETTEENTRY 对象数组中。
- UnrealizeObject 用于将指定的调色板与系统调色板相匹配。不带任何参数，功能与 Windows API 中相应的函数一致。
- ToClipboard 函数将调色板移入剪贴板。该函数带一个参数，即一个对 TClipboard 对象的引用。由于该函数会将调色板移入剪贴板，所以应该利用 TPalette 创建一个中间对象：

```
TClipboard clipBoard;
TPalette(tmpPalette).ToClipboard(clipBoard);
```

三、扩展 TPalette

Tpalette 函数有两个受保护的成员函数，都叫 Create。这两个函数的区别是：第一个 Create 函数的第一个参数是一个指向 BITMAPINFO 对象的指针，第二个 Create 函数的第一个参数是一个指 BITMAPCORE 对象的指针。Create 函数被 TPalette 的构造函数调用，TPalette 的构造函数带一个 BITMAPINFO * 或 BITMAPCOREINFO *，或 TDib& 作为参数。带 BITMAPINFO * 和 BITMAPCOREINFO * 的 TPalette 构造函数调用相应的 Create 函数。以 TDib& 为参数的 TPalette 构造函数从 TDib 对象中提取一个 BITMAPCOREINFO * 或 BITMAPINFO *，然后调用相应的 Create 函数。

两个 Create 函数的第二个参数均为一个 UINT 型数。该参数等同于 PALETTEEN-

TRY 结构的 peFlags 成员。peFlags 的合法值为 PC_EXPLICT,PC_NOCOLLAPS,PC_RESERVED 或者零。

Create 函数可根据传给它的位图头部包含的颜色表创建一个 LOGPALETTE 对象。该函数可用于2,16,25色位图。它不能用于其他类型的位图,包括24位的 DIB。然后调用 LOGPALETTE 创建一个 HPALETTE。

2.8.8 TBitmap 类

TBitmap 封装了 Windows 的设备相关位图,并提供了许多操作位图的构造函数及成员函数。

一、构造 TBitmap

构造 TBitmap 有两种方法,直接和间接。应用直接的方法,需要指定位图的宽度、高度等;应用间接的方法。需要指定一个已存在的位图对象、一个指向 BITMAP 结构的对象的指针,一个元文件、一个 TDC 设备上下文等信息。

下面的构造函数直接创建一个 bitmap 对象:

```
TBitmap(int width,int height,BYTE planes=1,BYTE
          count=1,void * bits=0);
```

width 和 height 定义的位图的宽度、高度(以像素为单位)。

Planes 定义位图的位平面数。count 定义每个像素所占的 bit 数。plane 或 count 必须为1,bits 是存储准备二进制信息的数组。如果 bits 为0,则表明位图还尚未初始化。

利用已有的位图可以创建一个位图对象。

```
TBitmap(const TBitmap& bitmap);
TBitmap(const BITMAP far * bitmap);
```

TBitmap 提供了两个构造函数,这两个构造函数可以创建与给定的设备上下文兼容的位图对象。第一个构造函数创建一个未初始化的位图,该位图的宽度为 Width,高度为 height,如果为 TBitmap 构造函数的 discandable 参数赋值 TRUE,则创建的位图可以丢弃;若位图被选入设备上下文,则该位图绝对不能丢弃。

```
TBitmap(const TDC& DC,int width,int height,
          BOOL discardable=FALSE);
```

第二个构造函数创建一个与某一设备兼容的位图,该设备由来自于 DIB 的设备上下文描述。usage 参数对于16位应用程序来讲,应当为 CBM_INIT,CBM_INIT 表示创建的位图需要由 DIB 对象包含的 Bit 信息初始化。如果不指定 CBMI_INIT,则创建一个空的位图。CBM_INIT 是创建位图时的缺省值。

二、存取 TBitmap

存取 TBitmap 需要借助于 HBITMAP 或 BITMAP 结构。利用 HBITMAP 操作符,将 TBitmap 对象作为参数,可得到该 TBitmap 对象的一个 HBITMAP。HBITMAP 类型转换操

作符是被隐式调用的：

```
HBITMAP GetHBitmap(TBitmap& bitmap)
{
    return bitmap;
}
```

上面的代码自动调用 HBITMAP 把 bitmap 转化为正确的类型。

利用 GetObjet 函数可以把 TBitmap 对象的转变为一个 BITMAP 结构：

```
BOOL GetBitmap(BITMAP far& dest)
{
    TBitmap bitmap(200,100);
    return bitmap.GetObject(dest);
}
```

GetObject 函数仅填充位图结构中的宽度、高度及颜色信息。利用 GetBitmapBits 函数可获取实际位图的 bit 信息。

TBitmap 封装了一些标准的 API 函数：

■ 下面的函数与 GetObject 函数返回相同的信息，每个函数位图的一个特定属性：

```
int Width()        BYTE planes()
int Height()       BYTE BitsPixel()
```

■ GetBitmapDimension 和 SetBitmapDimension 函数用于查询或改变位图尺寸。GetBitmapDimension 带一个参数，即一个对 TSize 对象的引用，它把位图的尺寸放入 TSize 对象中。SetBitmapDimension 带两个参数，第一个参数是一个对 TSize 对象的引用，该 TSize 对象包含设定的位图尺寸，另一个参数为一个指向 TSize 对象的指针，该对象返回旧的位图尺寸。若函数调用成功，返回值为 TRUE。

GetBitmapDimension 和 SetBitmapDimension 并不影响位图的像素尺寸。它仅影响位图的物理尺寸。在显示或打印时用到物理尺寸。利用这两个函数可根据物理屏幕调整位图大小。

■ GetBitmapBits 和 SetBitmapBits 用于查询和修改位图中的 bit 信息。两个函数都带有两个参数：一个 DWORD 和一个 void * ，DWORD 定义 void * 指向的数组的尺寸，（以字节为单位）。GetBitmapBits 函数利用 vodi * 指向的数组返回位图中的位信息。信息的量由 DWORD 参数确定。SetBitmapBits 与 GetBitmapBits 执行的功能相反，它将数组中的信息拷贝到位图中去。

■ ToClipboard 函数用于将位图移入剪贴板。它带一个参数，即一个对 TClipboard 对象的引用。由于 ToClipboard 将位图移入剪贴板，因此在移动之前应用 TBitmap 构造函数创建一个临时位图对象：

```
TClipboard ClipBoard;
TBitmap (tmpBitmap)ToClipboard(clipBoard);
```

三、扩展 TBitmap

TBitmap 有三个受保护(protected)成员,:一个构造函数和两个 Create 函数。

该构造函数是缺省的构造函数,利用它可构造一个派生类而避免直接显示地调用该函数。但在派生类的构造函数中,需要正确地初始化位图。

第一个 Create 函数有一个参数,即一个对 TBitmap 对象的引用。该函数对传递给它的位图进行自我复制,复制后的位图覆盖原来的位图。

第二个 Create 函数带两个参数,即一个对 TDib 对象的引用和一个对 TPalette 对象的引用。Create 首创建一个与 TPalette 对象兼容的设备上下文,然后将 Dib 包含的信息读入一个设备兼容的位图。

2.8.9 TRegion 类

利用 TRegion 可在设备上下文中定义一个区域,然后对之进行各种操作,例如喷漆,填充,反相等,这时设备上下文相当于一个模板。另外也可用 TRegion 定义一个区域进行自定义的各种操作。

一、构造和撤销 TRegion

利用 TRegion 定义的区域可以具有各种不同的形状,可以有不同的尺寸。从直角矩形到圆角矩形乃至多边形。即可以利用 TRegion 的构造函数直接构造一个具有指定形状的区域,也可基于一个区域句柄或现有的 TRegion 对象构造一个区域。

TRegion 提供了一个用于构造空的矩形区域的缺省构造函数。利用 SetRectRgn 可对空的区域进行初始化。例如,从 TRegion 类派生一个新类,在派生类的构造函数中调用 SetRectRgn 进行的初始化。这样可避免显示地调用 TRegion 的构造函数:

```
class TMyRegion:public TRegion
{
    pulbic:
        TMyRegion(TRect& rect);
          ⋮
};
TMyRegion::TMyRegion(TRect& rect);
{
    // Initialize the Tregion bas with rect
    SetRectRgn(rect);
}
```

有多种方法直接创建一个 TRegion 对外,下面的构造函数创建一个简单的矩形区域:

```
TRegion(const TRect& rect);
```

上面的构造函数利用 TRect 对象包含的逻辑坐标,创建一个矩形区域。

用下面的构造函数创建一个圆角矩形区域:

```
TRegion (const TRect& rect,const TSize& corner);
```

在上面的构造函数中，利用 TRect 对象包含的逻辑坐标创建一个直角矩形区域，然后根据 TSize 对象的信息对四角进行圆角化。

用下述构造函数可创建一个椭圆区域：

```
TRegion(const TRect& e,TEclipse);
```

创建的椭圆区域的矩形边界由 TRect 对象的确定。TEclipse 是一个枚举型数据，其值为 Ellipse。调用该构造函数的方法如下：

```
TRect rect(20,20,80,60);
TRegion rgn(rect,TRegion::Ellipse);
```

用下述构造函数可创建一个不规则多边形：

```
TRegion(const TPoint * points,int count,int fillMode);
```

其中，point 是一个 TPoint 对象数组，每个对象包含有欲创建的多边形的一个顶点坐标。count 定义了 points 数组中对象的个数，亦即多边形的边数，fillMode 确定对创建区域进行填充的模式，如 ALTERNATE 或 WINDOING。若创建一个具多个不规则的多边开的区域，可用下面的构造函数：

```
TRegion(const TPoint *points,const int * polycounts,
        int count,int fillMode);
```

其中，point 是 TPoint 对象数组，它存储了所有的多边形的顶点坐标。PolyCounts 存储了每一多边形的顶点数。count 确定了多边形的个数。fillMode 为填充模式：ALTERNATE 或 WINDING。

例如，欲创建一个封装了两个三角形区域的区域。由于每个三角形有三个点，所以 points 数组应有6个成员。polyCount 应有2个成员，其值均为3，因为是两个三角形。count 的值为2，表明该区域内有2个多边形。

基于现有的 HRGN 也可创建一个 TRegion 对象：

```
TRegion(HRGN handle,TAutoDelete autoDelete=NoAutoDelete);
```

另外，基于现有的 TRegion 对象也可创建一新的 TRegion 对象。

```
TRegion(const TRegion region);
```

TRegion 删除所创建的矩形区域并释放所占的内存空间。

二、存取 TRegion

利用一个 HRGN 句柄或一些 TRegion 成员函数及操作符可对 TRegion 对象进行存取和修改。使用 HRGN 操作符，把 TRegion 对象作为参数，可得 TRegion 对象的 HRGN 句柄。在这种情况下，HRGN 操作符是被隐式调用的：

```
HRGN TMyBitmap::GetHRgn()
{
```

```
    return *this;
}
```

上面的函数隐式调用类型转换操作符 HRGN 把 TRegion 对象转换为正确的类型——HRGN。

1. 成员函数

TRegion 提供了若干成员函数，利用这些函数可获取有关 TRegion 对象的信息，包含确定某一点是否在所创建的区域内：

■ SetRectRgn 函数设置对象的区域为一矩形区域：

```
void SetRectRgn(const Trect & rect);
```

该函数将 TRegion 对象设定为 rect 所包含的值。注意，该函数无条件地设置 TRegion 对象的形状为矩形，不管先前是何种外形。

■ Contains 函数用于确定某一点是否在一区域内：

```
BOOL Contains (const TPoint& point);
```

point 包含欲查询的点，若该点在区域内，Contains 返回 TRUE，否则返回 FALSE。

■ Touches 函数用于确定某一矩形是否包含在一区域内：

```
BOOL Touches (const TRect& rect);
```

rect 包含了欲查询的矩形坐标。若该矩形包含在区域内，Touches 返回 TRUE，否则返回 FALSE。

■ GetRgnBox 函数返回一边界矩形区域的坐标。

```
int GetRgnBox(TRect& box);
TRect GetRgnBox();
```

边界矩形是指可以包含某一区域的最小矩形。该函数有多个版本，第一个版本带一个参数，即一个对 TRect 对象的引用。该函数边界矩形的坐标放入 TRect 对象中。其返回值指明了区域的复杂程度，其返回值可为：SIMPLEREGION，COMPLEXRGN，NULLREGION。若函数调用失败，返回值为 ERROR。第二个版本的 GetRgnBox 不带任何参数，返回一个 TRect 对象，该对象包含了边界矩形的坐标。它不能返回区域的复杂程度。

2. 操作符

TRegion 提供了多个操作符。可用来查询，修改区域的值。大部分操作符还可用于往一区域中增加或撤销其他类型的区域。

TRegion 提供了两个布尔(Boolean)测试符：==和!=。用于比较两个区域是否相等，若相等==，返回值 TRUE，!=返回 FALSE 若两个区域不等，==返回 FALSE，!=返回 TRUE。这两个操作符也可用于比较 int，char 类型的数据。

例如，欲比较两个区域是否一致，若不一致，进行一定的处理。可用下述代码实现：

```
TRegion rgn1;
TRegion rgn2;
```

```
// initialize regions...
if(rgn1!=rgn2)
{
    // perform your operation here
    ⋮
}
```

TRegion 另外还提供了基本赋值操作符，它们用于改变区域的值：

- =操作符用将一个区域赋给另一个区域。例如，语句：rgn1=rgn2，将 rgn2的内容赋于 rgn1，而不管在此之前 rgn1的内容是什么。
- +=操作符移动一个区域，移动的位移由一个给定的 TSize 对象确定。该运算类似于数值加法。它将区域中的每个点都移动指定的位移。该区域仍保留原有特性，只不过是区域的坐标已经改变，变化量由 TSize 对象中的 cx 和 cy 成员确定：
  - 如果 cx 是正值，区域右移 cx 个像素。
  - 如果 cx 是负值，区域左移 cx 个像素。
  - 如果 cy 是正值，区域向下移动 cy 个像素。
  - 如果 cy 是负值，区域向上移动 cy 个像素。

  例如，欲将一个区域右移50个像素，向上移20个像素，实现代码如下：

```
TRegion rgn
// Initialize region...
TSize size (50,-20);
rgn+= size;
// continuing working with new region
⋮
}
```

- -=操作符与 TSize 对象连用时，执行与+=相反的功能。它把区域中的每一个点都减去给定的偏移量。例如，仍用上面的代码，只不过是+=变为-=。这时，它将会把区域左移50个像素，向下移20个像素。
- 当将-=操作符与 TRegion 对象连用时，执行的功能与上面介绍的不同。为了搞明白-=是如何工作的，请看下面的代码：

```
TRegion rgn1,rgn2;
        rgn1-=rgn2;
```

  执行了上述代码之后，rgn1区域为原来的区域除去 rgn2的区域，因此 rgn1与 rgn2为两个互不重合的区域。
- &=可以与 Tregion 对象的 TRect 对象一起使用(在进行操作之前，调用构造函数 TRegion::TRegion(TRect&)把 TRect 对象转换为 TRegion 对象)。下面的代码显示了 &=是如何工作的。

```
TRegion rgn1,rgn2;
        rgn1 &=rgn2;
```

执行了上述代码之后,rgn1的区域包含了原来与rgn2共有的区域。这是一个逻辑与操作:只有在执行逻辑与操作以前,两区域的共有部分才成为rgn1新的区域。

■ |=与&-一样,都可与TRegion对象或TRect对象一起使用。下面的代码演示了|=的使用方法:

```
TRegion rgn1,rgn2;
        rgn1 |=rgn2;
```

上述代码执行之后,rgn1的区域为其原有的区域加上rgn2的区域。这是个逻辑或(OR)操作:两区域相加组成新的区域。

■ ^ =也可以与TRegion对象或TRect对象一起使用。下面的代码演示了^ =的使用方法:

```
TRegion rgn1,rgn2;
        rgn1 ^ =rgn2;
```

执行上述代码后,rgn1的区域为:原rgn1区域加上rgn2的区域,再减去rgn1与rgn2共享的区域。即这实际上是一个逻辑异域(XOR)操作。

2.8.10 TIcon 类

TIcon类封装了一个图标句柄以及实例化图标对象的构造函数。利用该构造函数,可以从已有的资源构造出一个图标。

一、构造TIcon

构造TIcon对象的方法很多:利用现有的图标资源,利用应用程序中的资源、利用其他模块中资源、或者是确定的尺寸及数据信息来构造图标对象。

利用封装在TIcon对象中的现有图标创建图标对象:

```
TIcon(HINSTANCE instance,const TICon &. icon);
```

instance是一个模块实例。例如可以先从DLL中获取一个实例,然后从该实例中得到一个图标:

```
TModule iconLib("MYICONS.DLL");
TIcon icon cliconLib,"MYICON");
```

在调用TIcon的构造函数时,隐式地将TModule iconLib转换为一个实例句柄,HINSTANCE。

利用模块中已有的图标资源,创建一个TIcon对象:

```
TIcon(HINSTANCE instance,TResId resId);
```

这里,instance表示包含图标资源的模块句柄。resId为图标资源标识ID。如果欲使用Windowws内部提供的图标,为instance参数传递0值。

另外,也可从一个文件中装载一个图标:

```
TIcon (HINSTANCE instance,char far * filename,int index);
```

这里，instance 是当前模的句柄，filename 是包含图标的文件的名字。index 是欲装入的图标在文件中的索引。

利用下面的函数也可创建一个新图标：

```
TIcon (HINSTANCE instance,TSize& size,int planes,int bitsPixel,
       void far& anBits,void far * xorBits)
```

这里，instance 是当前模块的句柄，Size 是图标的尺寸，planes 是图标的位平面数。bitsPixel 是每像素的二进制位数、andBits 指向一个包含图标 AND 掩码的数组，xorBits 指向一个包含图标 XOR 掩码的数组。andBits 数组定义一个单色掩码，xorBits 数组可以是一个单色位图或设备相关彩色位图。利用已有的的 HICON 句柄也可创建一新的 TIcon 对象：

```
TIcon(HICON handle,TAutoDelete autoDelete=NoAutoDelete);
```

下面介绍两个仅适用于32位应用程序的构造函数：

```
TIcon(const void * resBIts,DWORD resSize);
TIcon(const ICONINFO * iconInfo);
```

第一个构造函数有两个参数：resBits 是一个指针，指向包含 Icon 数据位的缓冲区。resSize 定义了在 resBits 指向的缓冲区中图标数据位的数量。

第二个构造函数仅有一个参数：即一个 ICONINFO 结构。该构造函数利用 ICONINFO 结构包含的信息创建一个图标。ICONINFO 结构成员 fIcon 的值必须是 TRUE，表示该结构包含的是一个图标。

TIcon 删除图标，释放内存空间。

二、存取 TIcon

利用 HICON 句柄可以存取 TIcon。利用 HICON 操作符，把一个 TIcon 对象的作为参数，可以得到该对象的 HICON 句柄。在这里，HICON 操作符是被隐式调用的：

```
HICON TMyIcon::GetHIcon(
{
    return *this;
}
```

另外 GetIconInfo 函数也可用于 TIcon，但仅限于32位应用程序。它带一个参数，即一个指向 ICONINFO 结构的指针。该函数用图标信息填充这一结构，若操作成功，返回 TRUE。例如，先创建一个 TICON 对象，然后获取该对象信息，并放它放在 ICONINFO 结构中，可用如下代码：

```
ICONINFO iconInfo;
// Load stock icon-Exclamation
TIcon icon(0,IDI_EXCLAMATION);
icon.GetIconInfo(&iconInfo);
```

2.8.11 TCursor 类

TCursor 类封装了一个光标句柄，以及实例 TCursor 对象的构造函数。利用 TCursor 类，

即可根据光标资源,也可指定明确的信息来创建光标。

一、构造光标

构造光标有多种方法:从一个现有的 TCursor 对象,从当前应用程序中的资源,或确切的尺寸及数据信息来构造光标对象。

利用 TCursor 对象封装的光标创建一个光标对象:

```
TCursor(HINSTANCE instance,const TCursor& cursor);
```

instance 是当前应用程序的句柄。TCursor 没有封装应用程序实例,因为根本不知道应用程序对象。在一个窗口或其他界面对象中很容易存取当前应用程序实例。

```
TCursor (HINSTANCE instance,TResId resId);
TCursor(HINSTANCE instance,const TPoint& hotSpon,
        TSize& size,void far& andBits,void far * xorBits);
```

利用现有的 HCursor 句柄创建新的 TCursor 对象:

```
TCursor (HCURSOR handle,TAutoDelete autoDelete=NoAutoDelete);
```

Tcursor 删除光标对象,如果删除失败,析构函数将产生一个意外事件。

二、存取 TCursor

存取 TCursor 需借助于 HCURSOR 句柄,利用 HCURSOR 操作符,把 TCursor 对象作为参数,可以得到一个 HCURSOR 句柄。在这里,HCURSOR 操作符是被隐式调用的:

```
HCURSOR TMyCursor::GEtHCursor()
{
    return * this;
}
```

在上述代码中,隐式调用 HCURSOR 将 TCursor 对象转换为一个 HCURSOR 句柄。

TCurosr 提供了一个 GetIconInfo 函数,它仅适用于32位应用程序。该函数有一个参数,即一个指向 ICONINFO 结构的指针。在返回之前要填充该结构。如果调用成功返回 TRUE。例如,先创建一个光标对象,然后提取光标信息填充 ICONINFO 结构。代码如下:

```
ICONINFO cursorInfo
// LOad stock cursor-slashed circle
TCursor cursor(NULL,IDC_NO);
cursor.GetIconInfo(&cursorInfo);
```

2.8.12 TDib 类

设备无关位图,即 DIB,没有 GDI 句柄,仅有一个全局句柄。它只是一个包含了格式信息,调色板及位信息的结构。借助于 TDib 类,很容易使用 DIB,就像应用其他 GDI 对象一样简单。DIB 所需要的内存是调用 Windows 函数 GlobalAuoc 分配的,因此它可以被传递给剪贴板,OLE 服务器或客户机,以及其他的独立于应用程序的对象。

一、构造与删除 TDib

可以用直接或间接的方法构造 TDib。利用直接的方法，需要指明位图的宽度，高度等信息。利用间接的方法，需要指定一个已存在的位图对象，一个指向 BITMAP 结构的指针，一个元文件或一个 TDC 设备上下文等。

下面的构造函数用直接的方法构造一个 TDib 对象：

```
TDib (int width,int height,int nColors,WORD
        mode=DIB_RGB_COLORS);
```

width 是宽度，height 是高度，均以像素为单位。nColors 是 DIB 所用的颜色数。mode 的合法值为：DIB_RGB_COLORS 或者 DIB_PAL_COLORS。DIB_RGB_COLORS 表示颜色表中颜色的 RGB 值。DIB_PAL_COLORS 表示颜色表存储的一组16位的相对当前调色板的颜色索引。

另外，从可执行的应用程序模块中也可装入一个 TDib 对象，实现该功能的构造函数需要两个参数：第一个是包含现有位图的模块的句柄，第二个是欲装入的位图在该模块中的资源 ID：

```
TDib(HINSTANCE instance,TResId resId);
```

利用剪贴板创建一个 TDib 对象：为构造函数传递一个对 TClipboard 对象的索引，该构造函数先得到剪贴板中位图的句柄，然后利用该句柄创建一个位图对象。

```
TDib (const TClipboard& clipboard);
```

从文件中(尤其是.BMP 文件)装入 DIB。这时需要为构造函数提供文件名字：

```
TDib(const char * name);
```

给定一个 TBitmap 对象或一个 TPalette 对象也可创建一个 TDib 对象：

```
TDib(const TBitmap& bitmap,const TPalette & pal=0);
```

根据已有的 DIB 对象创建一个新的 DIB 对象：

```
TDib(const TDib& dib);
```

根据已有的 HGLOBAL 句柄创建一个新的 TDib 对象：

```
TDib(HGLOBAL handle,TAutoDelete autoDelete=NoAutoDelete);
```

由于 HGLOBAL 句柄可指向多种不同的对象，因此在应用它时，必须确保它是一个设备无关位图的句柄。

如果 ShowDelete 是 TRUE，~TDib 释放资源，解锁并释放分配的全局内存块。

二、存取 TDib

TDib 提供许多不同类型的函数用于存取封装的 DIB。

1. 类型转换函数

这些类型转换函数使存取位图变得非常方便。

利用 HANDLE 转换存取 TDib。把 TDib 对象作为参数，利用该转换符，可得到该 TDib 对象的 HANDLE 句柄。这里，HANDLE 转换符是被隐式调用的：

```
HANDLE TMyDib::GetHandle()
{
    return * this;
}
```

另外，还可将 TDbi 对象转换三种其他的位图类型：BITMAPINFO *，BITMAPINFOHEADER *，TRgbQuad *：相应操作符为：

```
operator BITMAPINFO far * ();
operator BITMAPINFOHEADER far * ();
operator TRgbQuad();
```

2. 存取内部结构函数

这里介绍的几个函数可用于存取 DIB 的内部数据结构：

```
BITMAPINFO far * GetInfoHeader();
TRgbQuad far * GetColors();
```

下面的函数返回一个指针，该指针指向存储 DIB 颜色索引的 WORD 数组。

```
WORD far * GetIndices();
```

下面的函数返回一个指针，该指针指向存储位图位信息的数组：

```
void HUGE * GetBits();
```

3. 剪贴板函数—— ToClipBoard

ToClipboard 函数可将一个 DIB 移入剪贴板。该函数带一个参数：即一个对 TClipboard 对象的引用。在执行该函数之前应先创建一个临时对象：

```
TClipboard clipBoard;
TDib(ID_BITMAP).ToClipboard(clipBoard);
```

4. 查询 DIB 信息

TDib 提供基本函数用查询 TDib 对象及其包含的信息：

- IsOK 函数检测 TDib 对象是否有效，若有效，返回 TRUE，否则，返回 FALSE。
- IsPM 函数不带任何参数。若指定的 DIB 是一个兼容的 Presentation Manager 位图。返回 TRUE，否则返回 FALSE。
- Width 和 Height 函数以像素为单位返回位图的宽度及高度。
- Size 函数以像素为单位返回位图的宽度、高度尺寸，该尺寸包含在一个 TSize 对象中。
- NumColors 函数返回位图中所用的颜色数。
- StatScan 总是返回零，是为了旧版本的代码兼容而提供的。

■ NumScans 是为了与旧版本的代码兼容而提供的，它返回位图的高度(以像素为单位)。

■ Usage 函数指出 DIB 的颜色模式：DIB_RGB_COLORS 或者 DIB_PAL_COLORS。

■ WriteFile 函数把 DIB 写入磁盘。该函数带一个参数，即 const char *，它指向欲用于存储 DIB 的文件名。

5. 工作于调色板或 RGB 模式

DIB 有两种方法保存其颜色值。在调色板模式下，DIB 的颜色表包含了颜色在调色板中的索引。在这句中情况下，颜色值本身并不代表具体的颜色，需要利用索引查找当前调色板才能得到特定的颜色，在 RGB 模式下，DIB 颜色表的每一项都包含颜色的 RGB 值。利用下面两个函数在两种模式间切换：

```
BOOL ChangeModeToPal(constTPalettes& pal);
BOOL ChangeModeTORGB(constTPalettes& pal);
```

当利用 ChangeModeToPal 函数切换到调色板模式时，TPalette& 参数用作 DIB 的调色板。DIB 中的颜色映射到调色板中并转换为相应的索引。当调用 ChangeModeToRGB 切换到 RGB 模式时，TPalette& 参数被用来把当前的调色板索引转换为等同的 RGB 值。

如果工作于 RGB 模式下，可用下列函数存取、修改 DIB 的颜色表：

■ GetColor 函数返回 DIB 颜色表中的某一项。该函数带一个参数，即一个索引用值，int 型。它返回一个 TColor 对象。

■ SetColor 函数用于设置 DIB 中颜色表的某一项。该函数带两个参数；一个 int 型参数，表示欲修改项的索引。一个 TColor 对象，包含欲设置的颜色值。

■ FindColor 函数带一个参数，即一个 TColor 对象。它遍历颜色表项，查找是否有与 TColor 对象相匹配的表项。若没有查到匹配的表项，返回-1。

■ MapColor 函数用于替换颜色表中存在的某一个颜色值。它带三个颜色：第一个是一个 TColor 对象，它包含了欲被替换的颜色值；第二个参数也是一个 TColor 对象，它包含欲进行替换的颜色值。第三个参数是一个 BOOL 量，它表示是否对颜色表中所有与第一个参数相匹配的表项都进行替换。如果该参数是 TRUE，则所有与第一个参数相匹配的表项都要用第二个参数进行替换；如果是 FALSE，则仅替换第一个匹配项。该参数的缺省值是 FALSE。

例如，欲进行全程替换、代码如下：

```
myDib->MapColor(TColor::LtGray,TColor::White,TRUE);
```

如果工作于调色板模式下，可用下列函数存取，修改 DIB 的颜色表：

■ GetIndex 返回颜色表中的某一项在调色板中的索引。它带一个参数，即一个指示颜色表索引的 int 型参数。返回一个 WORD 型数，它包含了调色板索引。

■ SetIndex 函数用于改变颜色表中的某一项。该函数带两个参数：一个 int 参数，欲修改的颜色表项的索引。一个 WORD 参数，对调色板的索引。根据该索引指定的颜色修改颜色表。

- FindIndex 用在颜色表中匹配某一个调色板索引。该函数仅有一个 WORD 参数。它在颜色表中查找与 WORD 参数指定的颜色相匹配的值。如果没有找到匹配的项，返回－1。
- MapIndex 函数用于替换在颜色表中存在的某一颜色。该函数带三个参数：第一个是一个 WORD 参数，欲被替换的调色板索引；第二个也是一个 WORD 参数，欲被替换的调色板索引；第三个参数是一个 BOOL 量，表示是否要对颜色表中所有与第一个 WORD 相匹配的项都用第二个参数进行。替换。如果该 BOOL 量是 TRUE，则全部进行替换；如果是 FALSE，则仅替换第一个匹配项。缺省条件下，该 BOOL 值为 FALSE。该函数的返回值表示实际进行替换的项数。

6. 匹配界面颜色与系统颜色

DIBs 经常用于美化用户界面。为了使用户界面系统一致，应当调用 MapUIColors 函数。该函数用系统颜色取代标准界面颜色。下面是调用 MapUIColors 函数的句法：

```
void MapUIColors(UINT mapColors,TColor * bkColor=0);
```

其中：mapColors 是下面几个标志的或(OR)组合：TDib::MapFace，TDib::MapText，TDib::MapShadow，TDib::MapHighlight 和 TDib::MapFrame。每个值都将引起不同的颜色替换，如下表所示：

This flag	Replace...	With....
TDib::MapText	TColor::Black	COLOR_BINTEXT
TDib::MapFace	TColor::LtGray	COLOR_BTNFACED
TDib::MapFace	TColor::Gray	COLOR_BTNSHADOW
TDib::MapFace	TColor::Write	COLOR_BTNHIGHLIGHT
TDib::MapFace	TColor::LtMagenta	COLOR_WINDOWFRAME

如果指定了 bkColor 参数，将会用 bkColor 替换 TColor::LtYellow

由于该函数寻找并替换 TColor 表项，因此该函数仅工作于 DIB 的 RGB 模式下。而且，由于它替换特殊的颜色，因此在设计界时要应用标准系统颜色。例如，按钮上的文本应当是黑色，按钮表面应是浅灰色(TColor::LtGray)。这样做很简单，因为它们都是特殊设计的，它们等同于标准的缺省用户界面元素的颜色。

三、扩展 TDib

TDib 提供了若干保护性的成员函数，仅能在 TDib 类中或 TDib 的派生类的使用。另外还可存取 TDib 中的一些控制数据：

- Info 是一个指向 BITMAPINFO 或 BITMAPCOREINFO 结构的指针。该结构包含了 DIB 的颜色表、属性、以及其他的信息。
- Bits 是一个 void 指针，它指向一个存储 DIB 图形数据的内存区。
- NumClrs 是一个 long 数，记录了 DIB 中实际应用的颜色数。
- W 是一个 int 数，表示 DIB 的像素宽度。
- H 是一个 int 数，表示 DIB 的像素高度。

- Mode 是一个 WORD 数，表示该 DIB 是属于 RGB 模式还是调色板模式。
- IsColor 是一个 BOOL，如果是 TRUE，info 指向一个 BITMAPCOREINFO 结构。
- IsResHandle 表示 DIB 是否是作为一个资源装入的。如果是按资源装入的，Handle 则是一个资源句柄。

InfoFromHandle 函数填充了 Info 指向的结构。该函数从 Handle 中提取信息，然后填充 Info 结构。它不带参数，也无返回值。

Read 函数从一个 File 对象指明的文件中读入 DIB。在装入时，Read 检查 DIB 的文件头，属性，调色板，位图。如果读操作作成功，返回值 TRUE。

LoadResource 函数从一个应用程序中或一个 DLL 模块中装载 DIB。该函数有两个参数：一个应用程序或 DLL 模块的句柄，一个资源标识 TResId 对象。如果操作成功，返回 TRUE。

LoadFile 从文件中装入 DIB。该函数有一个参数，即一个文件名字符串的指针。该文件包含了欲装入的 DIB。如果 LoadFile 执行成功，返回值 TRUE。

2.9 配件及配件窗口对象

这一节讲述配件和配件窗口的应用。从功能上讲，配件和控制相同，用来收集用户的输入或向用户传达信息，但配件和控制执行原理不同，和绝大多数界面元素不同，配件不是窗口，它没有窗口句柄，不能接收事件和消息，它们不是从 TWindow 派生出来的。

然而，配件只有在窗口中才能存在，且由窗口提供其显示，处理消息。配件从窗口接收命令和指令。

本节讲述了 ObjectWindows 2.0提供的各种配件，然后讲述配件窗口如何使用这些配件。

2.9.1 配件(Gadgets)

这一部分从配件的基类 TGadget 开始，讲述配件，然后讨论其他一些配件类：TSeparatorGadget，TBitmapGadget，TControlGadget，TTextGadget 和 TButtonGadget。

一、TGadget 类

所有配件都基于 TGadget，它包含所有配件所需的基本功能，如控制配件边框和边框类型，设置配件的大小，使配件使能/无能等

(一)构造和撤销 TGadget

TGadget 的构造函数如下：

```
TGadget(int id=0,TBorderStyle style=None);
```

这里：

- id 是配件的 ID 值，使用这个 ID 值来区分配件窗口中的配件，ID 值的其他用途后一部分来讨论。

■ style 是 TBorderStyle Enum 值。有五种可选值：

① None：表示配件没有边框，即，没有可见的边框。

② Plain：表示配件的边框看起来像一条线，很像窗口框架的边框。

③ Raised：表示配件看起来像凹进配件窗口中一样。

④ Recessed：表示配件边框看起来像凹进配件窗口中。

⑤ Embossed：表示配件边框看起来像有一个凸起的边。

TGadget 析构函数声明为 virtual，其唯一功能是当配件窗口合法时将配件从配件窗口中移走。

（二）区分配件

通过 GetId 函数读取标识符来区分一个配件，GetId 没有参数，返回一个标记符的 int 值，标识符的值是 TGadget 构造函数的第一个参数。

配件标识符有以下一些用途

■ 使用标识符区分一个具体的配件，如果配件窗口中有很多配件，区分这些配件的简单方法就是用标识符。

■ 如果配件是用来产生一个命令，就可以将配件标识符设为事件标识符，如用 button 配件打开一个文件，将配件标识符设为：CM_FILEOPEN。

■ 如果要在消息框或状态行显示配件按下时的字节，就可以将配件标识符设为字串的标识符。例如，如果一个字串的标识符为 IDS_MYSTRING，该字串描述了配件，那么就可以将配件标识符设为 IDS_MYSTRING。如果窗口有一个消息或状态条，而且打开了菜单跟踪，那么，无论何时按下配件 IDS_MYSTRING。字串 INS_MYSTRING 就会在消息行或状态行中显示出来。

后两种技术一般合二为一。假设有一个命令标识符 CM_FILEOPEN，用来执行菜单 File|Open 命令，你就可以将配件标识符设为 CM_FILEOPEN。这样，当按下配件时，配件窗口处理 CM_FILEOPEN 事件，同时，如果有一个字串，其资源标识为 CM_FILEOPEN，当按下配件时，该字串就会显示在消息行或状态行。

（三）修改和读取配件外观

可以修改和检测边缘宽度，边框的宽度，边框风格。函数如下：

```
void SetBorders(TBorder& borders);
TBorder &GetBorders();
void SetMargins(TMargins& margins);
TMargins &GetMargins();
void SetBorderStyle(TBorderStyle style);
TBorderStyleGetBorderStyle();
```

边框是指配件的最外层界限，SetBorders 和 GetBorders 函数用的 TBorder 结构有四个数据成员。这些 unsigned 数据成员：Left，Right，Top 和 Bottorn 包含了配件的宽度和高度。

niargin 是配件的边框和内层矩形之间的区域，SetMargins 和 GetMargins 函数使用的 TMargins 结构有四个数据成员。这些 int 成员 Left、Right，Top 和 Bottom 包含了配件的边缘宽度。

SetBorderStyle 和 GetBorderstyle 函数用 TBorderStyle 枚举和值 TGadget 构造函数的值相同。不同的边框效果通过绘制配件边框和边缘的边得到。

（四）定位 **gadget**

配件的最大矩形是一个配件占据的整个面积。其值包含在 TRect 结构中。由配件相对于配件窗口的左上和右下角坐标组成。配件窗口用这些约束矩形来设置配件。配件的约束矩形在用户单击配件时，用来检测到底单击了哪个配件。

查找和设置配件的约束矩形，使用以下的函数：

```
TRect & GetBounds();
virtual void SetBounds(TRect & rect);
```

注意：*SetBounds* 声明为 *virtual*，缺省的 *SetBounds* 只更新约束，矩形值。在派生类中重写 *SetBounds* 来监控配件的变化和更新配件的内部状态。

（五）折绕配件

可以用 SetShrinkWrap 函数指定配件窗口是否是"shrink wrap"配件，如果打开 shrink wrapping 标志，则 gadget 要求的全部大小自动根据边框大小，边缘大小，内部矩形的大小计算出来。这使用户不用手工计算配件的大小。

可以单独为高度和宽度打开 Shrink wrapping 标志。

```
void SetShrinkWrap(BOOL shrinkWrapWidth, BOOL shrinkWrapHeight)
```

这里：

- shrinkWrapWidth 打开水平折绕标志(on-打开，off-关闭)
- shrinkWrapHeight 打开垂直折绕标志。

（六）设置 gadget 尺寸

gadget 的尺寸是配件约束矩形的尺寸。它不依赖于配件的位置；因而，不用改变配件的位置就可以调整配件的大小。

使用 SetSize 函数设置配件的尺寸：

```
void SetSize(TSize & Size);
```

使用 GetDesiredSize 获取配件的尺寸

```
virtual void GetDesiredSize(TSize& size);
```

即使通过 SetSize 函数设置了配件的尺寸，仍然应该调用 GetDesiredSize 函数获取配件

的尺寸，在配件被安放其间，配件窗口有可能改变配件的尺寸。

(七)使 gadget 颜色和系统颜色匹配

为了使接口和应用程序的接口一致，应该写 SysColorChange 函数。当窗口接收到 WM_SYSCOLORCHANGE 消息时，配件窗口会为每一个配件调用 SysColorChange 函数，语法如下：

```
virtual void SysColorChange();
```

缺省的 SysColorChange 函数不做任何事，如果要使配件颜色跟随系统颜色而改变，就必须完成这个函数，必须删除和重新分配那些机系统颜色相关的资源。

(八)TGadget 的公有数据成员

TGadget 有两个 BOOL 公有数据成员：

```
BOOL Clip;
BOOL WideAsPossible;
```

Clip 表示在绘制配件时是否要应用剪接矩形；

WideAsPossible 表示配件是否要扩大以适应窗口中空间的大小。这在消息条中用文本配件(textgadget)时是很有用的。

(九)使配件有效或失效

用下列函数使配件有效/无效

```
virtual void SetEnabled(BOOL);
BOOL GetEnabled();
```

用函数 SetEnabled 改变配件的状态会使配件的约束矩形无效，但不会删除配件，派生类重写 SetEnabled 可改变这种行为，如果配件是用来产生命令的，就要完成 CommandEnable 函数。

```
virtual void CommandEnable();
```

缺省时，CommandEnable 不做任何事。派生类可以重载这个函数以提供命令。配件必须发送一个 WM_COMMAND_ENABLE 消息给配件窗口的父窗口。

CommandEnable 函数如下：

```
void TMyGadget::ComamndEnable()
{ Window->Parent->HandleMessage(
  WM_COMMAND_ENABLE,0,
  (LPARAM) &TMyGadgetEnabler( * Window ->Parent,this));
}
```

二、从 TGadget 派生类

TGadget 提供了几个保护函数可用于 TGadget 的派生类

（一）初始和清除函数

TGadget 提供两个 virtual 函数来初始化或清除配件。

```
virtual void Inserted();
virtual void Removed();
```

把一个配件插入配件窗后调用 Inserted 函数，而要将配件从配件窗口移走之前调用 Re-moued()函数，缺省时，这两个函数不做任何事。

（二）绘制 gadget

TGadget 提供两个不同的绘制函数：PaintBorder 和 Paint

PaintBorder 函数绘制配件的边框，这个 virtual 函数带一个 TDC& 参数，返回 void。paintBorder 执行标准的边框风格，如果要创建一个新的边框，必须重载这个函数，并提供创建新边框的代码，如果要提供标准的边框，调用 TGadget 这个函数。PaintBorder 函数被 Paint 函数调用。

Paint 函数和 TWindow::Paint 函数相似，该函数带一个 TDC& 参数，返回 void，Paint 声明为 virtual。当配件窗口绘制时，TGadget 函数 PaintGadget 调用每个 gadget 的 paint 函数，缺省的 Paint 函数只调用 PaintBorder 函数。如果要绘制配件约束矩形的内部矩形，必须重载这个函数的功能。

如果正在重绘配件，通常要知道边框和边缘之间的区域大小，这个区域称为内部矩形，使用 GetInnerRect 函数获取其大小：

```
void GetInnerRect(TRect& rect);
```

GetInnerRect 把内部矩形的坐标放在 Trect 引用中

（三）使配件无效及更新配件

如窗口一样，配件也可以使其无效，TGadget 提供两个这样的函数：

```
void Invalidate(BOOL erase=TRUE);
void InvalidateRect(const TRect& rect,BOOL erase=TRUE);
```

这些函数和 TWindow 数的 InvalidateRect 和 Invalidate 函数相似。InvalidateRect 和 Windows API 的函数很相似，只是没有 HWND 参数，Invalidate 使整个配件的约束矩形无效，它带一个 BOOL 参数，表示更新时无效区域应该被删除。缺省时，该参数为 TRUE，表示清除整个区域，调用 Invalidate 时，或者指定 TRUE 参数，或者不指定参数。

相关的函数是 Update 这个函数更新配件，它和 Windows API 函数 UpdateWindow 相同。

```
void Update();
```

（四）gadget 中的 Mouse 事件

可以在 gadget 中跟踪 mouse 事件，这通过定义在 Gadget 类中的一系列“伪事件处理

器"实现。这些函数和标准的 ObjectWindows 事件处理函数很相同,只是名字没有前缀 Ev。

gadge 没有响应表,这是因为 gadget 实际上不是窗口的缘故,所有的 gadget 只有通过配件窗口和外界通信,当一个 mouse 事件发生在 gadget 窗口中时,窗口就会决定哪个 gadget 受到了影响。如果要确定一个事件是否发生在一个 gadget 中,可以调用 PtIn 函数。

```
virtual BOOL PtIn(TPoint& point);
```

缺省时,如果 point 在 gadget 的约束矩形内,该函数返回 TRUE,如果你定义了一个稀奇古怪的 gadget,你必须重写这个函数。

当 mouse 进入 gadget 的约束矩形时,gadget 窗口就会调用 MouseEnter 函数。该函数如下:

```
virtual void MouseEnter(UINT modKeys,TPoint& Point);
```

modkeys 包含虚拟按键信息,和 ObjectWindows 的 EvMouseMove 函数中的信息相同,表示是否按下了某个虚拟键,这个参数是以下值的任意组合:MK_CONTROL,MK_LBUTTON,MK_MBUTTON,MK_RBUTTON 或 MK_SHIFT。point 告诉 mouse 在配件中的位置。

一旦配件窗口调用 gadget 的 MouseEnter 函数通知 gadget 鼠标已经进入了其区域,gadgett 就会调用 gadget 窗口的 GadgetSetCapture 函数捕捉 mouse 的运动,以保证 gadget 的 MouseLeave 函数被调用。

一旦 Mouse 离开了 gadget 按钮,gadget 窗口就会调用 MouseLeave 该函数,如下所示:

```
virtual void MouseLeave(UINT modKeys,TPoint& point);
```

有两个函数检测左 mouse 按钮:LButtonDown 和 LButtonUp。缺省时如果设置了 TrackMouse 标志,LButtonDown 捕捉 mouse,LButtonUp 释放 mouse 缺省时,TrackMouse 没有设置。

```
virtual void LButtonDown(UINT modkeys,TPoint& point);
virtual void LButtonUp(UINT Modkeys,TPoint& proint);
```

如果 mouse 移进了 gadget 的约束矩形,而同时 mouse 的运行正被 gadget 窗口捕获,则窗口调用 gadget 的 MouseMove 函数该函数如下:

```
virtual void MouseMove(UINT modKeys,TPaint& paint);
```

2.9.2　ObjectWindows gadget 类

objectwindows 提供了一系列 TGadget 的派生类,这些类提供了功能强大,且易使用的装饰件,并提供与应用程序 通信的方法。

这些 ObjedtWindows 类有:

- TSeparatorGadget
- TTextGadget
- TButtonGadget
- TControlGadget

■ TBitmapGadget

一、TSeparatorGadget 类

TSeparatorGadget 类是个很简单的 gadget，它的唯一功能就是在 gadget 窗口占一个位置，如果不想把配件边靠边放在窗口中，可以利用该类提供一个多余空间。

TSeparatorGadget 的构造函数如下：

```
TSeparatorGadget(int size=6);
```

分隔件关闭了 shrink wrapping 标志，且是无效的(disabled)。size 是宽度和高度的值。这可以让用户用分隔件在水平和垂直方向留出空间。

二、TTextGadget 类

TTextGadget 用来在 gadget 窗口中显示文本信息，你可以指定想在 gadget 中显示的字符数，以及文本的对齐方式。

(一)构造和撤销 **TTextGadget**

TTextGadget 的构造函数如下：

```
TTextGadget(int id=0,TBorderStyle style=Recessed,
            TAlign alignment=Left,UINT numchars=10,
            const char *  text=0);
```

这里：

- id 是 gadget 标识符
- style 是 gadget 边框风格
- align 指定文本在 gadget 中的对齐方式，有三种取值：

  TAlign:left,Center,和 Right

- numChars 指定显示的字符数目，这个数目规定了 gadget 的宽度。gadget 将字符数和当前字体的最大宽度相乘便得到所须的 gadget 宽度。gadget 的高度由当前字体的最大高度加上边框和边缘的空间。
- text 是显示在 gadget 中的缺省消息。

~TTextGadget 自动删除 gadget 文本字串的空间

(二)读取 gadget 文本

用 GetText 和 setText 函数设置 gadget 中的文本。

GetText 不带参数，返回 constChar *。不能通过返回的指针来试图修改 gadget 文本。

SetText 函数带一个 const char * 参数，返回 void。

gadget 作一个文本的拷贝并存起来

三、TBitmapGadget 类

TBitmapGadGet 是一个简单的类，主要用来显示 bitmap 数组，每次一个位图。必须把

bitmap 作为一个数组存起来。要做到这一点，必须在一个单独的 bitmap 资源中画出这些位图(side by side)每个位图必须有相同的宽度。

(一)构造和撤销 **TBitmapGadget**

TBitmapGadget 的构造函数如下：

```
TBitmapGadget(TResId bmpResId,int id,
              TBorderStyle style,int numImages,int startImage);
```

这里：

- bmResId 是 bitmap 资源的资源标识符
- id 是 gadget 标识符
- Style 是 gadget 边框风格
- numImages 是位图中的图像总数，gadget 将 bitmap 的宽度除以 numImages 得到单个图像的宽度。

  例如，向 TBitmapGadget 构造函数传送一个400像素宽200像素高的位图资源，指定 numImages 为4，这样，构造函数将 bitmap 资源分为4个单独的位图，每个位图为100像素宽200像素高。
- startImage 指定位图存 bitmap 的存储区。

(二)选择新的图像

调用 SelectImage 函数来改变显示 gadget 中的位图。

```
int SelectImage(int imageNum,BOOL immediate);
```

imageNum 是欲在 gadget 中显示的位图索引，如果为 immediate 赋值 TRVE，则会立即更新显示，否则，区域只是无效。当下一个 WM_PAINT 消息来时，才会更新。

(三)设置系统颜色

TBitmapGadget 完成了 SysColorChange 函数以便位图跟踪系统颜色，它删去位图数组，调用 MapUIColors 函数，然后重新创建数组。有关 MapUIColors 函数的更多信息，参见后续有关章节。

四、TButtonGadget 类

TButtonGadget 是 objectwindows 中唯一可以和用户直接交互的 gadget 控制(后一部分将讨论)，也提供了接收用户输入的功能，但它是通过控制类进行的。这种情况下，gadget 只是 gadget 窗口和控制之间的中介。

button gadget 有三种常规状态：UP Down 和 Indeterminate。另外，在三种状态下按 button 时，button 都会高亮显示。.

有两种 button gadget：命令配件和设置配件，设置配件是互斥的(像 radio button)或非互斥的(像 check box)。命令配件只能在"up"状态。设置配件可以在三态。

当 mouse 指针位于配件的约束矩形内，且左按钮按下时，button 配件就被按下。button 配件按下时，会高亮显示。

一旦 button 配件被按下，它就会捕捉 mouse 的移动。当 mouse 移出 gadget 的约束矩形之外，但此时 mouse 左按钮仍然没有释放，则按钮高亮状态消失，但 gadget 继续捕获 mouse 的移动，如果 mouse 重新移进 gadget 的约束矩形，gadget 又会重新高亮显示。

当 mouse 左按钮释放后，不再捕获 mouse 的移动，当光标位于约束矩形内时，gadget 窗口将 gadget 标识符作为一个命令消息处理。

(一)构造和撤除 **TButtonGadget**

TButtonGadget 的构造函数如下：

```
TButtonGadget(TResId bmpResId,int id,
              TType type=Command,BOOL enabled=FALSE,
              TState state=Up,BOOL repeat=FALSE;
```

这里：

- bmpResId 是 button 中显示的位图的资源标识符，位图的大小。
  决定 gadget 的大小，因为打开了 shrink wriapping。
- id 是 gadget 标识符，这也是按下 gadget 时要处理的命令。
- type 规定 gadget 的种类。TType 有三种可能的值。

  ① Command 说明 gadget 是一个命令 gadget
  ② Exclusive 说明 gadget 是互斥的设置按钮，互斥按钮一般紧密相连，一同工作。可以通过插入其他的 gadget 设置互斥组，如 separator gadget 或 text gadget。
  ③ NonExclusiue 说明 gadget 是非互斥的设置按钮。

- enabled 指定第一次创建时，按钮 gadget 是否有效。
  如果创建按钮时，相应的命令有效，则按钮自动有效。
- state 是 buttongadget 的缺省状态 TState 有三种值：
  Up，Down 或 Inderminate(不确定)
- repeat 指定当按钮一直按住时，button 是否重复。如果 repeat 为 TRUE，则当按钮按下并被保持时，按钮重复。

~TButtonGadget 函数删除位图资源，如果资源信息包括在字串中，则删去字串。

(二)读取 **buttongadget** 的信息

有几个函数可以用来获取 button gadget 的信息。这些函数可以设置 gadget 的状态为合法 TState 值，获取 button gadget 的状态，获取 button gadget 的种类。

用 SetButtonState 设置 button gadget 的状态。

```
voit SetButtonState(TState);
```

用 GetButtonState 函数获取 button gadget 的当前状态

```
TState GetButtonState();
```

用 GetButtonType 函数获取 button gadget 的种类

(三)设置 button gadget 风格

使用以下函数修改 button gadget 的外形

■ 用 SetNotchCorners 打开/关闭 corner notching

```
void SetNotchCorners(BOOL notchCorners=TRUE);
```

■ 用 SetAntialiasEdges 将 antialiasing 关闭/打开

```
void SetAntialiasEdges(BOOL anti=TRUE);
```

■ 用 SetShadowStyle 函数改变按钮阴影的风格,有两种网络的按钮风格。TShadowstyle:SingleShadow 和 DoubleShadow。

```
void SetShadowStyle(TShadowStyle style=DoubleShadow);
```

(四)使命令有效

TButtonGadget 重载了 TGadget 函数 CommandEnalbe 用它来初始 WM_COMMAND_ENABLE 消息。

TButtonGadget::CommandEnable 函数如下:

```
void CommandEnable();
```

(五)设置系统颜色

TButtonGadGet 完成了 SysColorChange 函数以便 gadget 位图跟踪系统颜色。它用系统颜色重建 gadget。如果系统颜色发生了变化,则新的 ButtonGadget 就会反映这些变化。

五、TControl Gadget 类

TControlGadget 类用作 Window 控制(如 button ,editbox,listbox)和 gadget 窗口之间的接口,可以在 gadget 窗口中使用 Windows 控制,如 controlbar,statusbar 等。

(一)构造和撤销 TControlGadget

TControlGadget 的构造函数如下:

```
TControlGadget(TWindow& control,TBorderStyle style=None);
```

这里:

■ control 是指向 ObjectWindows 窗口对象,这个对象必须是合法构造的控制对象。

■ style 是 gadget 风格

~TControlGadget 函数撤除控制界面元素,然后删去控制对象的存储空间。

2.9.3 配件窗口

配件窗口基于类 TGadgetWindow，而 TGadgetWindow 又从 TWindow 派生，gadget 窗口拥有很多 gadget，在另一个窗口显示这些 gadget。

gadget 窗口为它包含的 gadget 提供很多功能。由于 gadget 不是窗口，它们不能处理或接收消息，或直接同窗口打交道，或为自身调用 Windows 函数。gadget 必须通过 gadget 窗口才能完成一些任务。gadget 几乎不能控制其在 gadget 窗口中的位置。gadget 窗口负责安放在一布置所有的 gadget。一般配件都安放五条线上，或水平或垂直。

gadget 窗口一般不能独立存在，它常包含在另外一个窗口中。gadget 窗口的父窗口一般是修饰的框架窗口，如 TDecoratedFrame 或 TDecoratedMDIFrame。TToolBox 常用 TFloatingFrame。

一、构造和撤销 TGadgetWindow

TGadgtWindow 的构造函数如下：

```
TGadgetWindow(TWindow * parent,
TTileDirection direction=Horizontal;
TFont * font==new TGadgetWindowFont,
TModule * module==0);
```

这里：

- parent 是指向父窗口对象的指针
- direction 是 enum TTileDirection。有两种取值：Horizontal 和 Verticat
- font 是 TFont 对象指针，这是 gadget 窗口包含的的缺省时，设为 TGadgetWindowFont，这是一种变宽度的 sans_Serif 字体，通常是 Helvetica。
- moudule 传给 TWindow 的基类 TModule 缺省时为0。

~TGadgetWindow 函数删去 gadget 窗口中的所有 gadget，然后删除字体。

二、创建 gadget 窗口

TGadgetWIndow 重写了 TWindow 的函数成员 Create()。该函数根据以下几个标准选择初始尺寸：

- shrink wrapping 是否由窗口中的任何 gadget 打开
- 窗口中所含的 gadget 的尺寸。
- 在 gadget 窗口中布置的方向
- gadget 是否有边框，以及边框的尺寸

Create 函数根据以上因素决定窗口的合适尺寸，设置窗口尺寸属性，然后调用基类 TWindow::Create 实际创建窗口界面元素。

三、将 gadget 插入 gadget 窗口

要使 gadget 窗口有效，必须包含一些 gadget，调用 Insert 函数将 gadget 放在 gadget 窗

口中。

```
virtual void Insert(TGadget& gadget,RPlacement
                    placement=After,TGadget * sibling=0);
```

这里:

- gadget 是需要插入 gadget 窗口中的 gadget。
- placement 指示 gadget 插入的位置。两个 TPlacement enum 值:Before 和 After,如果 sibling 参数指定了一个姐妹 gadget,那么根据 TPlacement 的值,将 gadget 插在该姐妹的 Before 或 After。如果 sibling=0,那么当 placement 为 Before 时,gadget 放在窗口中已有 gadget 的前面,当 placement 为 After 时,放在其后面。
- sibing 是指向姐妹的指针。

如果已经创建了 gadget 窗口,那么,调用 Insert 之后,必须调用 LayoutSession 函数,只有当窗口布置好以后,插入的 gadget 才会出现。

四、从 gadget 窗口移走 gadget

调用 Remove 函数从 gadget 窗口中移走一个 gadget。

```
virtual TGadget * Remoue(TGadget& gadget);
```

这里,gadget 是要移走的配件。

该函数从 gadget 窗口中移走 gadget,返回一个指针 TGadget * 。

配件并没有被删除,如果 gadget 不在窗口中,函数返回0。

同 Insert 函数,如果 gadget 窗口已经创建,调用 Remove 函数之后必须再调用 LayoutSession,只有重新布置 gadget 窗口,移走的 gadget 才不会出现。

五、设置窗口边缘和布置的方向

可以在窗口创建前或后改变边缘和布置的方向

使用以下两个函数:

```
void setMargins(TMargins& margins);
virtual void SetDirection(TTileDirection direction);
```

这两个函数设置合适的数据成员,然后调用 LayoutSession 函数。

通过调用 GetDirection 函数判断 gadget 安放的方向。

```
TTileDirectionGetDirection() const;
```

六 、布置 gadget

要布置一个 gadget,①调用 LayoutSession 函数。②virtual void LayoutSession()③LayoutSession 函数缺省时检测窗口界面元素是否已经创建,如果没有创建,函数立刻返回,不做任何事,如果界面元素已经创建,则自动布置窗口。当界面元素创建之后,LayoutSession 安放 gadget;并使修改区域无效。

如果改变了边缘，插入或移走了配件，或配件配件、窗口改变了尺寸，都会激活 LayoutSession 函数。

布置 gadget 的工作实际由 TTileGadgets 完成：

```
virtual TRect TileGadgets();
```

该函数决定每个 gadget 占用的空间然后依次放置它们。它返回 TRect，该值包含用于由于放置 gadget 后被修改的 gadget 窗口的尺寸。

TileGadGets 调用 PositionGadget。这使派生类可以调整配件之间的空间，以完成用户自己的安放计划。

```
virtual void PositionGadget(TGadget * prefious, TGadget * next,
                            TPoint& point);
```

该函数带用 previous 和 next 指向的配件，计算配件之间所须空间，然后填充 point。如果水平放置，则相关尺寸放在 point. x 中，如果垂直放置，相关尺寸放在 point. y 中。

七、当 gadget 改变尺寸后，通知窗口

如果 gadget 改变了尺寸，则调用 GadgetChangedSize 通知 gadget 窗口。该函数如下：

```
void GadgetChangedSize(TGadget& gadget);
```

gadget 是改变了尺寸的配件，缺省时，该函数简单激活 LayoutSession

八、折绕(shrinkwrapping)一个窗口

使用 SetShrinkWrap 函数指定 gadget 窗口是否要"折绕"一个 gadget，折绕配件窗口和折绕配件有一些区别，当在某一个坐标上折绕一个配件窗口时，根据放在该坐标上所需配件的尺寸自动计算该坐标尺寸。

可以单独将高度和宽度的 shrink wrapping 标志打开/关闭

```
void SetShrinkwrap(BOOL shrinkWrapWidth,
                   BOOL shrinkWrapHeight);
```

这里：

- shrinkWrapWidth 根据传递 TRUE 或者 FALSE 打开/关闭水平 shrinkwrapping 标志。
- shrinkWrapHeight 根据传递的 TRUE 或者 FALSE 打开/关闭垂直方向的 Shrinkwrapping 标志。

九、读取窗口字体

用函数 GetFont 和 GetFontHeight 获取当前的字体和字体高度：

```
TFont& GetFont();
UINT GetFontHeight()connst;
```

十、为 gadget 捕捉 mouse

当 mouse 位于 gadget 约束矩形内，且左键按下，就会通知 gadget。如果要传送 mouse 移动的通知消息，就必须捕捉 mouse 的移动。

通过使用 GadgetSetCapture 和 GadgetReleaseCapture 函数来达到这些目的。

```
BOOL GadgetSetCapture(TGadget& gaddget);
void GadgetReleaseCapture(TGadget& gadget);
```

两个函数中的 gadget 指定窗口要为哪个 gadget 设置或释放捕捉。

GadgetSetCapture 返回的 BOOL 表明捕捉是否成功。

窗口中的 gadget 通过 gadget 的 Window 指针来调用这些函数。

十一、设置提示模式

使用 SetHintMode 函数为 gadget 设置提示模式：

```
void SetHintMode(THintMode hintMode);
```

THintMode 有三种可选的值：

提示模式	显示的模式
NoHints	不显示提示
PressHints	当按下 gadget 直到释放之前一直显示提示
EnterHints	当 mouse 进入 gadget 时，显示提示

使用 GetHintMode 函数得到当前显示模式；

```
THintMode GetHintMode();
```

另一个函数，SetHintCommand 函数，决定什么时候显示提示：

```
void SetHintCommand(int id);
```

gadget 通过指向 gadget 窗口的指针 Window 来调用该函数。

gadget 窗口本身也调用这个函数。本质上，SetHintCommand 模拟菜单选择，使按下一个 gadget 等同于选择一个菜单。

为了使 SetHintCommand 和标准的 ObjectWindows 类协同工作，必须做一系列工作：

- gadget 窗口的父窗口即装饰窗口必须有消息行或状态行，
- 提示必须位于框架窗口里
- 必须有字串资源，其标识符和 gadget 的标识符相同。

即：如果 gadget 的标识符为 CM_MYGADGET，则必须有一个字串，其标识符也为 CM_MYGADGET。

十二、遍历 gadgets

使用下面的函数搜索 gadget 窗口中的 gadget。

```
TGadget * FirstGadget() const;
TGadget * NextGadget(Tgadget& gadget)const;
TGadget * GadgetFromPoint(TP;int& point)const;
TGadget * GadgetWithID(int id)const;
```

- FirstGadget 返回窗口 gadget 链表中指向第一个 gadget 的指针。
- NextGadget 返回窗口的配件链表指向下一个 gadget 的指针。如果当前 gadget 是最后一个 gadget 则返回0;
- GadGetFromPoint 返回点 point 所在的 gadget 指针，如果 point 不在 gadget 中，则返回0;
- GadgetWithId 返回具有标识符 id 的 gadget;如果窗口中没有具有该标识符的 gadget，则返回0。

从 TGadgetWindow 派生

可以从 TGadgetWindow 派生自己的特殊 gadget 窗口。

TGadgetWindow 具有一些 protected 函数。

(一)刷新 gadget 窗口

和常见窗口一样，TGadgetWindow 完成了 Paint 函数

```
void Paint(TDC& dc,BOOL erase,TRect& rect);
```

Paint 函数将字体选入设备上下文然后调用 PaintGadgets 函数:

```
virtual void PaintGadgets(TDC& dc,BOOL erase,TRect& rect);
```

PaintGadgets 递归浏览窗口中的 gadget，要求重绘自身。

重载 PaintGadgets 来给窗口新增一些功能，如分隔线等。

(二)尺寸和内部矩形

使用 GetDesiredSize 和 GetInnerRect 函数得到总体尺寸(即容纳边框，边缘，最宽或最高的 gadget 的尺寸)和窗口内部矩形的尺寸和位置。

```
virtual void GetDesiredSize(TSize& size);
```

如果窗口要求折绕，则 GetDesiredSize 计算窗口要用来容纳边框或边缘最高或最宽的 gadget 的尺寸;如果窗口不要求折绕，GetDesiredSize 使用当前宽和高。结果放在 Size 中。

```
virtual void GetInnerRect(TRect * rect);
```

该函数计算边框和边缘内的区域，结果放在 rect 中。

可以重载 GetDesiredSize 和 GetInneRect 函数为自定义的 gadget 窗口多留一点空间，如果重写两个函数中的一个，则也必须重写另一个函数。

(三)布置的单位(unit)

可以在 gadget 窗口中使用三种不同的测量单位:

- 像素,它基于屏幕的单个像素
- 布置单位,这是一种逻辑单位,定义为将窗口字"em"分成8个垂直和8个水平段;
- 边框单位,基于窗口框架的厚度,一般等同于一个像素,但高于屏幕分辨率。

一般用布置单位,因为它们基于字体大小,不用担心在窗口大小或系统尺寸变化后,相应改变测量尺寸。

使用 UnitsToPixels 函数将布置单位转化成像素。

```
int LayoutUnitsToPixels(int units);
```

units 是要转换的布置单位尺寸,函数返回像素大小。

使用 GetMargins 函数可以将 TMargins 对象转换成实际的像素尺寸:

```
void GetMargins(TMargins& margins,int& left,int& right,
                int& top,int& bottom);
```

这里:

- margins 是欲转换的尺寸。这些尺寸的单位可能是像素,布置单位,或者边框单位。
- 转换的结果放在 Left,right,top,bottom 中

十二、消息响应函数:

TGadgetWindodw 捕获以下消息:

- WM_CTLCOLOR
- WM_LBUTTONDOWN
- WM_LBUTTONUP
- WM_MOUSEMOVE
- WM_SIZE
- WM_SYSCOLORCHANGE

相应消息响应函数也包含在 TGadgetWindow。

2.9.4 ObjectWindows 配件窗口类

objectWIndows 提供了一系列从 TGadgetWindow 派生出来的类。这些窗口许多显示和布置 gadget 的方法。ObjectWindows 包含的窗口类有:

- TControlBar
- TMessageBar
- TStatusBar
- TToolBox

下面讨论这些类

一、TControlBar 类

TControlBar 类的提供控制条的功能。我们在很多应用程序窗口的顶部可以看到"Tool-

bar"或"control bar"。我们可以在控制条中放任何的配件。TControlBar 的构造函数如下:

```
TControlBar(TWindow * parent=0,TTileDirection direction=Horizontal,
            TFont * font=new TGadgetWindowFont,TModule * modeule=0);
```

这里:

- parent 是指向控制条父窗口的指针
- direction 是一个 TTileDirectionenum 有两种可能的取值:

  Horizontal 和 Vertical.

- font 是指向 TFont 对象的指针,这是 gadget 窗口的字体缺省时,它是 TGadgetWindowFont,这是一种变宽度的 sans-serif 字体,通常是 Helvetica。
- module 是传给 TWindow 基类 TModule 的参数,缺省时为0。

二、TMessageBar 类

TmessageBar 类创建一个没有边框,和窗口一样宽的文本 gadget 的消息条,它水平定位在其父窗口的底部。

(一)构造和删除 **TMessageBar**

TMessageBar 的构造函数如下:

```
TMessageBar(TWindow * parent=0,TFont * font=new
            TGadgetWIndowFOnt,TModule * module=0);
```

这里:

- parent 是指向控制条父窗口的指针
- font 是指向 TFont 对象的指针。这是 gadget 窗口的字体,缺省时,其值为 TGodgetWindowFont,这是一种变宽度的 sans_serif 字体,通常是 Helvetica。
- module 是传给 TWindow 基类 TModule 的参数缺省时为0。

~TMessageBar 函数删除对象的文本存储区

(二)设置消息条文本

用 SetText 函数设置消息条 textgadget 的文本。void SetText (const char * text);
这个函数将导致 text 显示在消息条中。

(三)设置提示文本

SetHintText 设置菜单或命令项提示文本,这些文本显示在条中域里。

```
virtual void SetHIntText(const char * text);
```

如果传递0,则清除提示文本。

三、TStatusBar 类

TStatusBar 和 TMessageBar 大致相同,区别在于状态条有多种选择,如文本 gadget,为

键盘模式提示符(CapsLock,Insert 或 Ouerwrite)保留的空间。

(一)构造和删除 **TStatusBar**

TStatusBar 的构造函数如下:

```
TStatusBar(TWindow * parent=0,
        TGadget::TBorderStyle borderStyle=TGadget::Recessed,
        UINT modeIndictors=0,TFont * font=new TGadgetWindowFont,
        TModule * module=0);
```

这里:

- parent 是指向父窗口对象的指针
- style 是 TBorderStyle enum
- moduelndicators 表示控制条中显示的键盘模式。定义的 enum 类称 TModeIndicator 该参数提供以下合法的值:
  - ExtendSelection
  - CapsLock
  - NumLock
  - ScrollLock
  - Overtype
  - RecordingMacro
- font 指向 gadget 窗口包含的 TFont 对象
- mudule 是传给 TWindow 基类 TModule 的参数

(二)将 **gadgets** 插入状态条

TStatusBar 重写了缺省的 Insert 函数,缺省时,TStatusBar 析这个函数在已有的文本 gadget 之后,模式指示 gadget 之前增加新的 gadget。

可以给 Insert 函数传递一个指向已有 gadget 的指针,将该指针作为要插入的新 gadget。然而,不能在模式显示符后面插入一个 gadget。

(三)显示模式指示符

为了显示一个具体的模式,在构造状态条时,必须指定该模式。一旦模式指示符显示在状态条上后,就由用户修改它们。

TStatusBar 提供几个函数来修改模式指示符。

使用 SetModeIndicator 将模式指针符的状态改为任何合法的状态。

```
void SetModeIndicator(TModeIndicator indicator,BOOL state);
```

这里:

- indicator 是要设置的模式指示符,它可以是构造函数中合法的 ModeIndicator 值。
- state 是模式指示符的设置状态。

也可以用 ToggleModeIndicator 函数转换模式指示符。

```
void ToggleModeIndicator(TModeIndicator indicator);
```

indicator 可以是合法的 TModeIndicator 值。

(四)设置状态条的 gadget 之间的距离

使用 SetSpacing 函数设置指示符 gadget 之间不同的距离。

```
void SetSpacing (TSpacing& spacing);
```

这里,spacing 是指向 TSpacing 对象的引用,TSpacing 是定义在 TStatusBar 中的一个 struct。它有两个数据成员,一个是 TMargins::TUnits,称 Units,另一个是 int 值 Value。TSpacing 构造函数设置 Units 为 TMargins::LayoutUnits,Value 为0。

四、TToolBox 类

TToolBox 和其他的 ObjectWindows 的 gadget 窗口类不同。它将 gadgets 不安排在一条线上,而是安排在一个矩阵里,矩阵的所有列具有相同的宽度(和最宽的 gadget 同宽),所有的列同高(和最高的 gadget 同高)。

可以把 TToolBox 作为 TFlotingFrame 的客户窗口,这样就可以作一个调色板式的工具箱。参见 examples\ow1\owlapps\paint。

(一)构造和撤除 TToolBox

这里 TToolBox 的构造函数:

```
TToolBox(TWindow * parent,int numColumns=2,
         int numRows=AS_MANY_AS_NEEDED,
         TTileDirection direction=Horizontal,
         TModule * module=0);
```

这里:

- parent 是指向父窗口对象的指针;
- numColumns 是工具箱中 gadget 的列数;
- numRows 是工具箱中 gadget 的行数;
- direction 是 TTileDirection 枚举(enum)有两种可能的值:Horizontal 或 vertical。如果 direction 为 Horizontal,则从左上角到右下角布置 gadgets,每一行填满。如果 direction 为 Vertical,则从左上角开始,往下只有一列,就填满这一列。
- Module 是传给 Tmodule 的参数,缺省时为0。

numColumns 或 numRows 的值可以设为 AS_MANY_AS_NEEDED,但两个参数不能同时设置该值。当将其中一个量的值设为 AS_MANY_AS_NEEDED,工具箱计算出要分多少小块。例如:如果有20个 gadgets,需要4列,这栗就可以安装五行。

(二)改变工具箱的尺寸

使用 SetDirection 函数来交换工具箱的尺寸

```
virtual void setDirection(TTIleDirection direction);
```

这里:direction 是 enum TTileDirection 有两种可选的值:

Horizontal 或 Vertical

如果 direction 和当前工具箱的方向不同,工具箱交换行和列的值。例如,假如你有一个3行5列的工具箱,垂直安放。调用 SetDirection 将 directi;n 设为 Horizontal,工具箱交换行和列的值,变成5行3列。

2.10 Doc/View 对象

ObjectWindows 2.0提供 Doc/View 模块来组织和操作数据,该模块由以下三部分组成:

1. 文档对象。这个对象能包含很多不同类型的数据,并提供存取这些数据的方法。
2. 浏览器对象。该对象是用户和文档对象之间的接口,它控制数据如何显示以及用户和数据如何打交道。
3. 一个面向应用程序的文档管理器。该管理器维护协调文档对象和与之对应的浏览器对象。

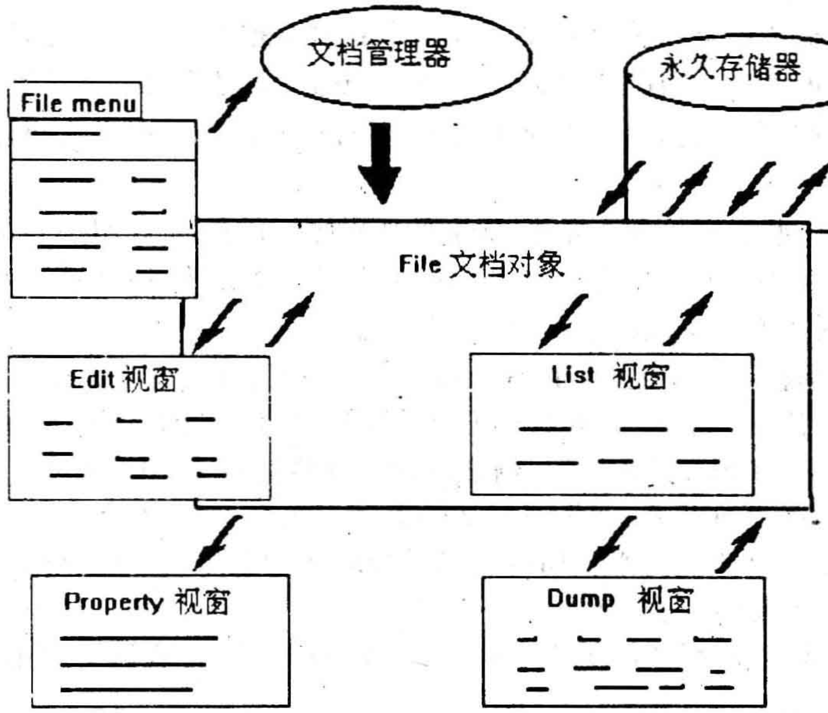
2.10.1 文档和浏览器如何协同工作

有了 Doc/View 模块,程序员就不用担心一个文件究竟由什么数据类型组成,以及这种数据如何在屏幕上显示,Doc/View 模块将文档类和浏览器类联系起来,文档管理器负责维护文档类型浏览器类之间的联系表。每一个联系称为一个文档模板(文档模板和 C++的模板没有联系)。

文档类处理数据存储和管理。它包含很多信息,这些信息在屏幕上显示出来。文档对象控制对数据的修改以及何时,以何种方式将数据传输并永久存储在介质里(如硬盘,RAM等)。

通过创建一个新文档或打开一个已经存在的文档,用户便打开了一个文档,此时该文档应用相关连的浏览器类将文档显示在屏幕上。浏览器类管理数据如何显示以及用户和屏显数据如何交互。实际上,浏览器在显示窗口和文档之间形成了一个接口。一些文档种类可能仅有一种相关的浏览类,另外一些种类则有几种相关的浏览类。不同的浏览器种类让用户以不同的方式与数据打交道。

下图表示了文档管理器,文档类以及和文档相关的浏览器是如何交互作用的。



这个图表表示的是一个文件文档对象，几个相关的浏览器。文档对象是类 TFileDocument 的对象。在\BC4\EXAMPLES\OWL\OWLAPI\DOCVIEW 中，有 TFileDocument 类的示例。这里 BC4是安装 Borand C++ 4.0的主目录。

一、文档

传统的文档概念和 Doc/View 文档的概念不同。传统的文档概念大体上类似于字处理文件的文档概念。它通常由文本组成，少数情况下还包括一些图形。一些内嵌的命令提供给字处理程序来处理文档的格式。Doc/View 文档和传统文档大相径庭，表现在以下几个方面：

1. 第一个区别是两种类型的文档内容不同。传统文档绝大多数是文本，极少情况下包括其他数据，而 Doc/Vieew 文档理论上可以包含任何种类的数据，如文本，图形，声音，多媒体文件，甚至其他的一些文档。
2. 第二个区别涉及到显示方式。传统文档的格式通常设计成和欲显示的方式保持一致，而 Doc/View 文档完全独立于显示方式。
3. 第三个区别是：某些字处理程序生成的文档通常依赖该字处理程序要求的格式。如果要在不同的字处理程序之间移植这些文档，则必须对文档进行繁琐的转换，而 Doc/View 文档可以在不同的应用程序之间方便的移植，即使这些应用程序的功能全然不同。

文档对象的基本功能由 objectWindows 类 TDocument 提供。有关如何使用 TDocument 的详细讨论，请阅读本章后续章节。

二、浏览器

浏览器对象保证将文档对象显示出来。如果没有浏览器对象，观察和修改文档对象是不可能的。将文档对象和浏览器对象结合成文档模板后，就能生成一块数据和代码，这些代码

将存储在文档中的数据图形化地显示出来,并提供与这些数据交互及修改这些数据的方法。

文档和流利器分隔开,也保证了在修改文档时的灵活性。尽管通过浏览器来操作数据,但是浏览器只是把这些更改传送给文档对象,由文档对象决定是否更改这些数据或者丢弃这些修改。

使用浏览器代替一些固定的显示方法(如字处理程序),还有另一个好处,那就是浏览器给程序员和用户提供了许多不同的方式来显示和操作同一个文档。尽管对一个文档只需提供一个浏览器,但有时人们还是想提供3个或4个浏览器。

例如,假设你想创建一个文档类存储图形信息,如一幅图片或者一张图,你可能只想提供一种类型的浏览顺,比如一个窗口,在这个窗口将图片显示,让用户涂抹或者修改图片。更进一步,你可能想提供一些额外的浏览器,例如,想提供浏览器显示图形的颜色;将图形作为十六进制文件显示出来;更有甚者,如果图形是用数学方式生成的,你完全可以用浏览显示这一串数学方程式。用户只要在打开文档时选择期望的浏览器类型,就可获取这些浏览器。所有这些方案中,文档本身绝不会变化。

浏览器对象的基本功能由 ObjectWindows 类 TView 提供。有关如何使用 TView 的详细讨论,请阅读后续章节。

三、关连文档类和浏览器类

通过文档模板,文档类便和相关的浏览器类联系起来。通过以下两步来创建文档模板:

1. 定义一个模板类来联系文档类和浏览器类;
2. 实例化模板类

这两步之间的区别是很重要的,应特别注意。当定义一个模板类后,就可创建这个模板类的任意数量的实例。每个模板只和一个文档类和浏览器类联系。每个实例有一个名字,缺省的文件后缀。目录、标志以及文件过滤符。所以,只要提供单个的模板类来联系一个文档和一个浏览器,然后就可以创建该样板类的许多实例,每个实例就能用相同的文档类和浏览器类处理不同缺省目录下的文件,这些文件可以带不同的扩展名。

四、管理 Doc/View

文档管理员负责维护应用程序的模板实例列表和当前文档的列表。每个应用程序,只要使用 Doc/VIew,必须有一个文档管理员,而且同一时刻,只能有一个文档管理员。

文档管理员协调文档类,浏览器类和模板之间的关系,以便 Doc/View 模块顺利工作。文档管理员提供一个缺省的文件菜单以及菜单中各个选项的缺省处理,如下:

菜单选择	缺省处理
New	创建一个新文档
Open...	打开已经有的文档
Save	存储当前文档
As...	用新名字丰当前文档
Revert To Saved	将文档恢复未存储之前的状态
Close	关闭当前文档
Exit	退出应用程序,提示存储文档

一旦完成了文档类和浏览器类，定义了必须的模板，并实例化了模板，剩下要做的便是创建文档管理器，文档管理器来设置模板实例的链表和菜单（如果定义了构造函数）。之后，只要接到待处理的事件，它就调用相应的命令来完成。如何建立文档管理器，请看后续例子。

2.10.2 文档模板

文档模板创建一个新类将文档类和浏览器类连在一起。文档管理员则维护文档模板的链表。当创建新的 Doc/View 实例时，必须用到这些文档模板，这一部分说明如何创建和使用文档模板。

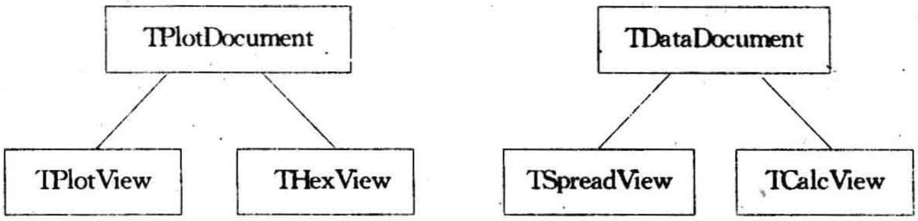
一、设计文档模板类

使用宏定义 DEFINE_DOC_TEMPLATE_CLASS 来创建文档模板类，这个宏定义有三个参数：

- 文档类
- 浏览器类
- 模板类名

文档类必须是能作为数据包容的文档类；浏览器类则能用来显示文档类中的数据；模板类名最好能反映该模板的功能，但不能是 C++关键字（如 int，switch 等），也不能是应用程序中其他项的名字。

下面举例说明。假设有两个文档类，一个是 TPlotDocument 它含有图形数据；另一个称为 TDataDocument，它包含数字数据，现在假设有四个浏览器类，每个文档类有两个浏览器类。对 TPlotDocument 类，一个是 TPlotView 类，用来将 TplotDocument 对象中的数据作为图形显示出来；另一个是 THexView 类，用来将 TPlotDOcument 对象中的数据作为十六进制数字序列显示。对 TDataDocument 类，一个是 TSpreadView 类，用来将 TDataDocument 中的数据以表格的形式显示出来；另一个是 TCalcView，用来处理 TDataDocument 对象中的数据，然后显示处理后的数据，上述关系见下图：



为了联系文档类和浏览器类，可以利用宏定义，写出如下的代码：

```
DEFINE_DOC_TEMPLATE_CLASS(TPlot Document,TPlotView,TPlotTemplate);
DEFINE_DOC_TEMPLATE_CLASS(TPlotDocument,THexView,THexTemplate);
DEFINE_DOC_TEMPLATE_CLASS(TDataDocument,TSpreadView,'tspreadTemplate);
DEFINE_DOC_TEMPLATE_CLASS(TDataDocument,TCalcView,TCalcTemplate);
```

以第一行为例，已知的文档类 TPlotDocument 和浏览器类 TPlotView 通过一个新的模板类 TPlotTemplate 联系起来，其他三行雷同。于是，创建了四个新的模板类：TPlotTemplat，THexTemplate，TSpreadTemplate 和 TCalcTemplate。下一部分讲述如何使用这些新建的模

板类。

二、创建模板类实例

一旦定义了模板类，就可创建该模板类的任意个实例。应用模板类实例，可提供一个模板的不同的描述方式，搜索不同的缺省文件名，在不同的目录中查找等等。这些工具都能在调用模板类构造函数时完成。

模板类的命名总是相同的：

```
TplName name(LPCSTR desc,LPCSTR filt,LPCSTR dir,
                LPCSTR ext,Long flags);
```

这里：

- TplName 是定义模板时的模板类名。
- name 是该模板类的一个实例名
- desc 是有关模板的描述文字
- filt 是一个或多个合法的正则表达式，之间用分号隔开；它实际上是一个字符串，用来过滤当前目录下的文件名；
- dir 是搜询文档文件的缺省目录；
- ext 是存文件时的缺省后缀，为0时无缺省后缀。
- falgs 是文档打开或创建时的模式，可以是下述一个或多个组合：

Flag	描　述
dtAutoDelete	当上一个浏览器关闭时，自动关闭和撤销文档对象
dtNoAutoView	不能自动创建缺省的浏览器
dtSingleView	每个文档只能有一个浏览器
dtAutoOpen	创建时自动打开文档
dtHidden	从用户选项列表中隐藏模板

例如，假设有下面的模板定义：

```
DEFINE_DOC_TEMPLATE_CLASS(TPlotDOcument,
                          TPlotView,TPlotTemplate);
```

现在假定要创建这个模板类的三个实例：

- 一个实例具有描述文字"Approved plots"，文档文件具有后缀.PLT，且该文件位于目录 E:\APPROVED 下。该文档只有一个浏览器，且当浏览器关闭时自动撤销文档。
- 另一个实例具有描述文字"In Progress"，文档文件具有后缀.PLT，位于目录 E:\WORK 下。当最后一个浏览器关闭时自动撤销文档。
- 最后一个实例具有描述文字"Propsals"，文档文件具有后缀.PLT 或.TMP(缺省时为.PLT)，位于目录 E:\TMP 下。只有当用户输入一个通行字后该模板才出现，当最后一个浏览器关闭时撤销文档对象。

创建这些实例的代码如下：

```
TPlotTemplate atpl("Approved plots"," * . PLT","E:Approved\SAMPLE",
                    "PLT",dtSingleview|dtAutoDelete);
TPlotTemplate btpl("In progrress"',"*.PLT","C:\WORK",
                    "PLT",dtAutoDelete);
TPlotTemplate *ctpl=new TPlotTemplate("Proposal","*.PLT","*.TMP","C:\tmp",
                                       "PLT",dtHidden|dtAutoDelete);
```

像其他类一样,同样可以创建模板类的静态如动态实例。

三、修改模板类

一旦创建了一个模板类的实例,一般没有必要修改模板对象。然而,某些情况下,还是想修改构造模板的一些特性,可以调用以下函数来做这些事情。

■ 调用 GetFileFilter 和 SetFileFilter 来获取和修改当前目录下文件名的过滤符。
■ 调用 GetDescription 和 SetDescription 来获取和设置模板类的描述文字。
■ 调用 GetDirectory 和 SetDirectory 来获取和设置缺省目录。
■ 调用 GetDefaultExt 和 SetDefaultExt 来获取和设置缺省的文件后缀名;
■ 调用 GetFlags,IsFlagSet,SetFlag 和 ClearFlag 来获取和设置标志项。

2.9.3 使用文档管理器

文档管理器,作为 TDocManager 或其派生类的实例,完成以下任务:

■ 管理当前文档和注册过的模板;
■ 处理标准文件菜单命令:CM_FILENEN,CM_FILEOPEN,CM_FILESAVE,CM_FILESAVEAS,CMFILECLOSE,和 CM_FILEREVERT。
■ 提供文件选择接口。

为了支持 Doc/View 模板,文档管理器必须附着在应用程序里。这可以通过创建 TDocManager 的实例并使它成为应用程序的文档管理器来完成。以下代码示例如何将管理器附着在应用程序里:

```
class TMyApp:public TApplication
{ public:
      TMyAppl();TApplication() {}
      void InitMainWindow(){
       ⋮
      SetDocManager(newTDocManager(dmMDI|dmMenu));
       ⋮
      }
  };
```

用 SetDocManager 函数将文档管理器设置为一个新的对象。SetDocManager 带一 TDocManager& 参数,返回 void。

通过调用文档类的 GetDocManager 能够存取文档管理器的公共数据和函数。GetDoc-

Manager 无参数，返回 TDocManager&。文档管理器提供下列函数创建文档和浏览器。

- TDocTemplate 的成员函数 CreateDoc 只提供它本身的模板；而 CreateAnyDoc 提供所有可见模板。
- CreateAnyView 从当前模板列表中过滤出支持当前文档的浏览器，并提供这些浏览器的名字列表；TDocTemplate 成员函数 CreateView 直接根据文档模板类定义来构造浏览器。

具体的文档管理器可以用作它用。如 OLE 2.0服务器需要支持一些类库；这些类通过并不属于自身的接口来创建文档和浏览器。如果用嵌入的命令行标志来激活服务器，它不会产生自己的用户接口，并能附着在一个文档管理器上，该文档管理器用适当的 OLE 支持来接替接口。

四、构造文档管理器

文件管理器构造函数带有一个参数，该参数设置文档管理器的模式。可以用两种模式创建文档管理器。

- 单文档接口模式 SDI，在这种模式下，任何时候，只有一个文档是打开的。如果你试图打开一个新文档，而此时另一个文档早已经打开，文档管理器就会关闭这个打开的文档，再创建新文档。
- 多文档接口模式 MDI。此时，可以打开多个文档及其浏览器。每个浏览器包含在它自己的应用程序窗口内，而且，每个文档可以是一种文档类型，用同一个浏览器显示，或者用不同的浏览器显示。有时文档类型甚至可以全然不同。

在 SDI 模式下打开文档管理器，带 dMSDI 参数调用构造函数，同样，在 MDI 模式下，带 dMMDI 参数调用构造函数。

还可以指定其他三个参数：

- dmMenu 参数指明文档管理器应该安装文件(File)菜单，该菜单提供标准的文档管理器 File 菜单项和相应的命令；
- dmSaveEnabled 参数。该参数保证 File 菜单上存盘命令 Save 即使在文档没有修改时仍然有效。
- dmNoRevert 参数使 File 菜单上 Revert 命令无效。

一旦构造了文档管理器后，模式就不能再更改。下列说明了如何创建两种模式的文档管理器，程序使用命令行让用户决定文档管理器是在 SDI 模式还是 MDI 模式上创建。

```
class TMyApp:pbulic TApplication
{ public:
    TMyApp():TApplication(){}
    void InitMainWindow();
    int DocMode;
};
void TMyApp::InitMainWindow()
```

```
{ switch((_argc>1 &&_argv[1][0]=='-'?_argv[1][1]
                                        :(char)0)|('S' '^' ' ' s'))
   { case 's':  DocMode=dmSDI;break;   // commamd line:-s
     case 'm':  DocMode=dmMDI;break;   // command line:-m
     default:   DocMode=dmMDI;break;   // no command line
   }
   SetDocManager(newTDocManager(docMode|dmMenu));
};
```

如果用-s 选项，则文档管理器以 SDI 模式创建，反之。用-m 选项或不带选项，文档管理器以 MDI 模式创建。

五、TDocManager 事件处理

如果创建 TDocManager 时带有 dmMenu 参数，则文档管理器负责处理某些事件。文档管理器通过应用响应表来处理标准的菜单命令，不管有没有文档打开，是否给应用程序增加资源，文档管理器都会提供这些菜单命令。File 菜单也是文档管理器提供的。

文档管理器处理事件有：

CM_FILECLOSE	CM_FILESAVE
CM_FILENEW	CM_FILESAVEAS
CM_FILEOPEN	CM_VIEWCREATE
CM_FILEREVERT	

有些情况下，你可能想处理这些事件。由于文档管理器的事件表是最后一个被搜索的表，所以你可以在浏览器，框架或应用程序级上处理这些事件。还有一种选择就是创建文档管理器时不带 dmMenu 参数，此时，必须提供函数来处理这些事件，一般是通过应用程序对象或者界面对象。

仍然可以通过应用程序对象的 DocManager 成员来调用文档管理器的函数。例如，在打开文件前执行一些任务。定义的窗口 TMyWindow 如下：

```
class TMyApp:public TApplication
{ pbulic:
    TMyApp():TApplication() {}
    void InitMainWindow();
    int DocMode;
};

void TMyApp::InitMainWindow()
{ // don't specify dmMenu when constructing TDocManager
    SetDocManager(new TDocManager(dmMDI));
}

class TMyWindow: public TDecoratedMDIFrame
{ public:
    TMyWindow();
```

```
    void CMFileOpen();
    DECLARE_RESPONSE_TABLE(TMyWindow);
};
DEFINE_RESPONSE_TABLE1(TmyWindow,TDecoratedMDIFrame)
    EV_COMMMAND(CM_FILEOPEN,CmFileOpen);
.
.
.
END_RESPONSE_TABLE;
void TMyWIndow::CmFileOpen()
{ GetApplication()->GetDocManager()->CmFileOpen();
}
```

2.10.3 创建文档类

文档类的基本任务是为浏览器里的数据变化提供回调函数;通过相关浏览器处理用户行为;当数据已经更新时通知浏览器。TDocument 类提供这些功能。编程人员只需给文档模块的具体应用程序增加所需的部件。

一、构造文档类

TDocument 是一个抽象基类,因而不能实例化。必须从 TDocument 类派生出一个类。

构造 TDoucment 的派生类时,必须调用 TDocument 构造函数。TDocument 的构造函数只带一个参数:即指向 TDocument 的指针 TDocument *。该指针指向新建文档的父类。如果新建文档没有父类,则传参数0或不传递参数,缺省值为0。

二、增加文档功能函数

要避免重载 TDocument 中未声明为 virtual 的函数,文档管理器在寻址所有 TDocument 派生类时,实际上把这些派生类对象作为 TDocument 类对象来寻址,所以如果重写了非虚拟函数,文档管理器并不会调用这个派生类的重载函数,相反它仍然调用基类 TDocument 的该函数。如果在派生类中重载了虚拟函数,文档管理器会正确的调用派生类中的重载函数。

在 TDocument 中,以下函数声明为虚函数:

~TDocument	SetDoc Path
InStream	SetTile
OutStream	GetProperty
Open	IsDirty
Close	IsOpen
Commit	CanClose
Revert	AttactStream
RootDocument	DetachStream

可以重写以上函数,提供自己的函数版本。但在重写这些 virtual 函数之前,必须清楚基类中这些函数的功能。如果基类的函数完成一些必不可少的功能,则应该在自己的重写函数中调用基类的该函数来完成那些功能,然后再写自己的代码。基类的很多函数要完成文档等级结构的检测,包括检测或证实任何子文档,所有浏览器,任何打开的流等。

数据存取函数

TDocument 提供很多函数来存取数据,可以把数据作为简单的流存取;或者用派生类中设计的任何方式进行存取。下列部分讲述一些有用的函数,当文档试图进行数据存取操作时,可以用这些函数进行控制。

(一)流存取

TDocument 提供 InStream 和 OutStream 两个函数,这两个函数分别返回 TInStream * 和 TOutStream * 指针。在 TDocument 中,由于这两个函数并没有做任何工作,所以返回零。如果要为自己的文档类提供流存取,则必须重写这两个函数,构造合适的流类,然后返回指向流对象的指针。

TInstream 和 TOutStream 都是抽象流类,分别从 TStream 和 istream 或 osstream 中派生而来。TStream 提供最小的功能将流和文档联系起来。istream 和 ostream 都是标准的 C++ iostreams。应该从 TiniStream 和 TOutStream 类派生出面向文档的流类。下面是一个有关 InStream 和 OutStream 成员函数的描述。InStream 和 OutStream 的构造函数都带两个参数。

```
XXXStream(int mode,LPCSTR strmId=0);
```

这里,XXX 代表 In 或 Out,mode 是流打开模式,及 istream 和 ostream 使用的 open_mode_Mode 标志相同,strmId 指向已有的流对象,传递一个合法的指针给一个已经存在的流对象,以使该流作为文档类的流对象,否则,就会创建一个新的流对象。

还有两个流存取函数,AttachStream 和 DetachStream。这两个函数均带一个指向已经存在的,从 TStream 派生出来的引用对象。AttachStream 将 TStream 的派生类对象加到流类对象的文档链表中,使其可以存取。DetachStream 函数则在流类对象的文档链表中搜索 TStream 派生类对象,并删除之。这两个函数均是保护成员,因而只能被文档对象调用。

(二)流链表

一个文档维护着打开流的链表,当流增加和删除时,链表随着更新。这个链表由 TDocument 类的数据成员 StreamList 打头。StreamList 实际上是一个 TStream * 指针,指向流链表中第一个流。如果流链表中没有流,则 StreamList 为0。链表中的每个 TStream 对象有一个 NextStream 成员,它指向下一个流。

如果文档对象里新建一个流,或者附着一个已有的流,它们都被加到流链表中。若流对象关闭一个流或解除一个流,这个流就会从流链表中删除。

(三)复杂的数据存取

流只提供简单方法来存取数据。有时候,文档包含多媒体文件,数据库表,或其他的一些复杂数据,此时就涉及到复杂的数据存取方法。鉴于此,TDocument 提供两个存取函数:

Open 和 Close。可以重写这两个函数，以便定义自己的打开和关闭代码。

在 TDocument 类中，Open 不 做任何事，且总是返回 TRUE。可以写出自己的 Open 函数来做想做的事情，没有限制条件，繁简均可。Open 打开一个文档，并保持一直打开。

TDocument 类的 Close 函数比 Open 多做一点事。它检测所有已有的文档子类，并在文档关闭之前关闭这些子类。如果要定义自己的 Close 函数，在该函数中最先要做的是调用 TDocument 的 Clos 函数，以保证父类文档关闭时，所有子文档都已关闭。除此限制之外，再没有其他限制。可以自由地定义 Close 函数的功能。像 Open 函数一样，在想关闭文档时，Close 也会关闭它。

(四)数据存取帮助函数

TDocument 类提供大量函数来帮助保护数据：

IsDirty 首先检查文档自身是否"脏"(即修改过，但还没有更新)，这是通过检测数据成员 DirtyFlag 的状态实现的，然后检测任何一个子文档是否"脏"，接着检测每一个浏览器是否"脏"，只要有一个子文档或浏览器是"脏"的，IsDirty 便返回 TRUE。

IsOpen 检测文档是否打开，或流链表中是否有流存在。如果文档没有打开，IsOpen 返回 FALSE，否则返回 TRUE。

Commit 执行对存储数据所作的任何修改。一旦调用了 Commit 函数，数据的任何修改都不可能再恢复成原样。TDocument 的这个函数检测所有的子类文档，然后执行对它们的修改，如果任何一个子文档返回 FALSE，则 Commit 失败退出，返回 FAISE，所有的子文档都必须返回 TRUE，这样 Commit 函数才会执行父类文档的数据更新。在子类文档都返回 TRUE 之后，Commit 函数刷新所有的浏览器，这些浏览器从上一次文档检测以来，有可能作了某些操作。按照浏览器里变化，文档里的数据被更新，然后存起来 Commit 返回 TRUE。

Revert 函数正好和 Commit 函数相反。Revert 函数不更新和存储数据，相反，它清除从上一次数据被 Commit 以来所做的任何修改。Revert 同样检测所有的子文档，如果任何一个子文档返回 FALSE，Revert 函数退出。只有所有操作都成功之后，Revert 才返回 TRUE。

三、关闭文档

如同其他对象，TDocument 提供函数安全地关闭和撤销文档对象。

~TDocument 做清除工作。首先，清除子类文档，关闭所有打开的流和资源；接着，按顺序撤除附属的模板，关闭所有关联的浏览器，删除流链表，然后从父类的子链表中移走(如果文档有一个父类的话)，或者从文档管理器的文档链表中移走(如果文档没有父类的话)。

除了析构函数，TDocument 还提供 CanClose 函数束保证正确关闭。CanCloset 先检测是否所有子类可以关闭，如果其中一个孩子返回 FALSE，则 FlushDOC 和 CanClose 返回 TRUE；如果文档不干净，则 FlushDoc 打开一个消息框提示用户是存盘还是放弃修改，还是取消关闭操作。

四、扩充文档的功能

本部分描述的函数包罗了绝大多数。你可以扩充文档类的功能。这个文档类必须有特殊的函数操作数据。理解和响应从与文档关联的浏览器获得的信息等所有这些功能都包括在

TDocument 的派生类中。

由于 Doc/View 模块相当灵活，所以在完成上述任务时，没有什么要求和规则。文档几乎可以处理任何种类的数据，这是因为 Doc/View 数据处理机制只是基本的框架结构，在派生类中功能可以扩展。ObjectWindows 的基类提供了扩充 Doc/View 模块的功能函数。

五、文档管理器

TDocument 提供两个函数操作文档管理器：GetDocManager 和 SetDocManager。GetDocManager 返回指向当前文档管理器的指针。利用这个指针可以操作文档管理器的数据和函数成员。SetDocManager 将文档管理员设为另一个文档管理员。所有其他功能都包含在文档管理器里。

六、浏览器

TDocument 提供两个函数来操作浏览器：NotifyViews 和 QueryViews。两个函数都带三个参数：int，是一个事件；long 型和一个指针 TView *。long 项依赖于事件，本质上是指向事件的参数；TView * 将浏览器排除在查询或通知之外，这只需将一个指针传送给这个浏览器即可。这两个函数是文档和浏览器之间传递消息的主要工具。

两个函数均通过浏览器的响应表来调用浏览器。ObjectWindows 通知事件使用的一般宏定义是 EV_OWLNOTIFY。EV_OWLNOTIFY 的响应函数如下的形式：

```
BOOL FnName(long);
```

NotifyViews 或 QueryViews 使用的 long 项用作该响应函数的 long 参数。

使用 NotifyViews 通知所有的子文档及其相关的浏览器，根文档的浏览器的数据发生了变化或被更新了，或其他需要在屏幕上作出反映的事件。事件的含义以及随参数传递给事件的项都是定义的执行项。

NotifyViews 首先调用所有文档的子文档的 NotifyViews 函数。待所有的子文档调用之后，NotifyView 传递事件及其项给所有文档的相关浏览器。NotifyViews 返回 BOOL 值。如果任意一个子文档返回 FALSE，则 NotifyViews 返回 FALSE。反之，返回 TRUE。

QueryViews 发送文件及相伴参数。QueryViews 和 NotifyViews 区别在于：当所有子文档或浏览器返回 TRUE 时，NotifyViews 才返回 TRUE，而 QueryViews 返回指向第一个返回 TRUE 的浏览器，这使得用户能够快速找到满足某种条件的浏览器，然后在这个浏览器中完成某种工作。如果没有浏览器返回 TRUE，QueryViews 返回0。另一个区别是：NotifyViews 将事件及其参数送给所有的子文档及其相关的浏览器，而 QueryViews 在第一个返回 TRUE 的浏览器处便不再发送消息及其参数。

举例说明。如果一个文档包含位图的数据，想知道与该文档相关的哪个浏览器正显示当前位图的某个区域。可以定义一个 WM_CHECKRECT。然后定义一个 TRect 结构包含想查找的矩形的坐标，其代码如下：

```
DEFIEN_RESPONSE_TABLE(TMyView,Tview)
EV_OWLNOTIFY(WM_CHECKREST,EvCheckRest),
```

```
.
.
END_RESPONSE_TABLE;
void MyDocClass::Function()
{ // set up TRect * with the coordinations you want to send.
  TRect * rect=new TRect(100,100,300,300);
  // queryviews
  TView * view=QueryViews(WM_CHECKRECT,(long)rect);
  // clear all changes from the view
  if(view)
    view-> Clear();
}
// THe view response function gets the pointer to the rectangle
// as the long paramenter to its response function.
BOOL TMyView::EvCheckRest(long item)
{ Trect * rect=(TRect * )item;
  // check to see if rect is equal to this view's
  if( * rect==this->rect)
      return TRUE;
  esle
     return FALSE;
}
```

2.10.5 创建浏览器类

用户一般不直接和文档打交道,相反,用户经常和接口对象如窗口,对话框,适合显示数据的任何显示以及处理数据的方法打交道。但这种接口对象不可能独自起作用,窗口对它显示的数据,对包含数据的文档,对用户如何操作和修改数据其实一无所知。所有这些功能都必须由浏览器对象处理。

浏览器在接口对象(该对象只能做别的对象叫它做的事情)和文档(该对象不知道如何叫接口对象做什么)之间形成了一个接口。浏览器的功能就是在两个之间搭起一座架,它从文档对象中读取数据,然后叫接口对象如何显示数据。

这一节讨论如何写一个浏览器类配合文档类工作。

一、构造 TView

不能直接创建 TView 的实例,TView 包含很多纯虚拟函数和"Place"函数,这些函数必须在派生类中提供。但在构造 TView 的派生类,又必须调用 TView 的构造函数。TView 带一个指向文档的引用。用户必须提供合法的 TDocument 派生类对象。

二、为浏览器增加功能

TView 包含一些纯 Virtual 函数,在任何一个新浏览派生类中,必须提供代码。它还包含

几个“placeholder”函数。如果要用这些函数，也必须有这些函数的新版本。

很像 TDocument 类，除非该函数是一个虚拟函数，否则不能重写 TView 的函数。当 TDocument 中的函数调用自定义浏览类中的函数时，它把自定义浏览当作是 TView 类。当重载一个非 Virtual 函数，并且文档调用该函数时，文档实际调用的是 TView 中的该同名函数，结果是用户定义的函数并没有执行。

(一)Tview 的 virtual 函数

下列函数声明为 virutal 函数，所以可以重载它们。但大多数并没有被声明纯虚拟(pure virtual)函数，所以在构造浏览器时，不一定非重载它们不可。只存在需要这些函数提供某些功能时，才重载它们。

GetViewName 返回 View 的名字，这个函数声明为 pure Virtual，必须在自定义浏览类中重载它。GetWindow 返回 TWindow *，如果浏览器有相关的接口对象，该参数指向该接口对象，否则，函数返回0。

SetDocTitle 设置浏览器的窗口标题，在接口对象中应该调用该函数。

(二)增加菜单

Tview 有一个 TmenuDescr * 数据成员 ViewMenu。可以将任何已知的 TMenuDescr 对象赋给这个成员。一般在 View 的构造函数中设置菜单，当 View 构造函数中设置菜单，当 View 激活时，该菜单和框架窗口菜单混合成一体。

三、给浏览器增加显示功能

TView 本身没有显示数据的功能，因为它没有指向窗口的指针，图形函数，文本显示函数，及键盘处理功能。必须在派生类中提供这些功能，可以用下列方法之一：

- 在派生类中增加一个指向接口对象的指针。
- 在派生新类时，将接口对象的功能和 TView 的功能结合起来。

这些方法有各自的优缺点，下一部分讨论这些问题。在创建新类时，应该权衡一下两种方法。

(一)增加接口对象指针

在 TView 派生类中增加指向接口对象的指针，增加新成员，并在 TView 派生类的构造函数中实例化该接口对象、通过对象指针来操作接口对象的数据成员和函数成员。

该方法的优点是用户能简单地将不同种类的接口对象附加或不附加在浏览类中；也可以使用接口对象的不同类型，只要创建一个指向这些对象的基类的指针即可，如通过创建指向 TWindow 的指针，就能使用很多种的接口对象。

这种方法不足之处是，事件处理必须首先遍历接口对象或应用程序，这就能使你要么使用派生的接口对象类增加事件处理函数，要么通过应用程序对象来处理事件；这样就降低了处理事件方面的灵活性。

(二)将 TView 和接口对象接合起来

将 TView 或 TView 派生对象和接口对象类结合起来,能获得显示文档的数据的能力,并且能将这种能力与处理出入文档对象的数据的流程集成一体。将浏览器类和接口对象类结合是一项很直接了当的任务,但必须很小心。

定义基于基类(TView 或 TView 派生类)的新类,选择接口对象。新类的构造函数必须构造两个基类,并初始化需要设置的数据成员。同时,新类必须定义那些在基类中声明为 Pure virtual 的函数,那些需要完成屏幕操作的函数以及定义事件处理函数来和接口对象元素文档对象通信。

该方法的优点是:事件处理放在中心位置,降低了在应用程序级处理事件的要求,不需要通过指针遍历接口元素的控制,接口元素的控制可以集成在新创建的浏览器类中。

但是,用这种方法失去了灵活性,不能将新的接口对象快速地附着或断开;接口对象是整个浏览器对象一个"器官",也不能通过指向不同接口对象类的基类指针来交换对象的不同类型。新创建的浏览器类被锁在接口元素的单个种类里。

(三)关闭浏览器

像其他对象一样,TView 提供函数来安全关闭和撤销对象。

~TView 做很少量的工作。它调用相关文档的 DetachView 函数,然后将自己从浏览器的文档链表中移走。

TView 还提供 CanClose 函数,该函数调用相关文档的 CanClose 函数。因而浏览器的关闭信号于文档的关闭。

2.10.6 Doc/View 事件处理

一般要通过应用程序对象和浏览器界面元素来处理 Doc/View 事件,既可以通过指向界面对象的指针来控制浏览器的显示,也可以将界面对象的功能和浏览器类混合起来。

在第五章中,可以找到更多有关事件处理和响应表的信息。

一、在应用程序对象处理 Doc/View 事件

应用程序对象通常只处理很少量的事件,如文档或浏览器的创建和撤除。无论何时创建浏览器或文档时,都要处理 dnCreate 事件,而当关闭文件或浏览器时,要处理 dnClose 事件。在响应表中增加宏 EV_OWLDOCUMENT 和 EV_OWLVIEW,为这些事件建立响应表入口项。

■ 使用 EV_OWLDOCUMENT 宏来检查

- 当新文档对象被创建时的 dnCreate 事件。处理事件的函数的标准名字是 EvNew-Document,该函数带一个 TDocument 派生对象的引用作参数,返回 void。
- 文档对象关闭时的 dnClose 事件。处理事件的函数的标准名为 EuCloseDocument,该函数带一个 TDocument 派生对象的引用作参数,返回 void。

这两个宏的响应表入口项和函数声明如下:

```
DEFINE_RESPONSE_TABLE1(MyDVApp,TApplication)
      EV_OWLDOCUMENT(dnCreate,EvNewDocument),
      EV_OWLDOCUMENT(dnClose,EvCloseDocument),
      .
      .
      .
   END_RESPONSE_TABLE;
   void EvNewDocument(TDocument& document);
   void EvCloseDocument(TDocument& document);
```

- 使用 EV_OWLVIEW 宏来检查。
- 构造新浏览器对象时的 dnCreate 事件。该事件处理函数为 EvNewView，该函数的参数是新创建的 TView 派生类对象的引用，函数返回 void。
  如果浏览器包含一个窗口界面元素，不论是通过继承还是通过指针，当浏览器创建时，界面元素还没有创建。在真正调用 Create 函数前，可以修改界面元素的生成属性。
- 撤销浏览器时的 dnClose 事件。该事件的标准处理函数为 EvCloseVIEW，该函数的参数是正被删除的 TView 派生对象的引用。函数返回 void。

这两个宏的响应表入口和函数定义如下：

```
DEFINE_RESPONSE_TABLE1(MyDVApp,TApplication)
   EV_OWLVIEW(dnCreate,EvNewView),
   EV_OWLVIEW(dnClose,EvCloseView),
   .
   .
   .
END_RESPONSE_TABLE;
void EvNewView(TView &view);
void EvCloseView(TView &view);
```

二、View 的 Doc/View 事件处理

头文件 docview.h 提供了很多预定义事件的响应表宏及事件处理函数，函数声明的类型检查等。用户也可以定义自己的事件和函数来处理用 NOTIFY_SIG 和 VN_DEFINE 宏定义的那些事件。

(一)处理预定的 **Doc/View** 的事件

有很多预定义的 Doc/View 事件。每个事件有相应的响应表宏定义和处理函数。注意 Doc/View 模块并没有提供这些函数，必须在自己的 View 类中声明这些函数并提供合适的代码。

下表是 Doc/View 的预定义事件：

响应表宏定义	事件名	事件处理	事　件
EV_VN_VIEWOPENED	vn-ViewOpened	VnViewOpened(TView *)	表示创建了新的 View
EV_VN_VIEWCLOSED	unViewClosed	VnViewClosed(TView *)	表示将关闭 view
EV_VN_DOCOPENED	unDocOpened	VnDocOpened(int)	表示已经创建了文档
EV_VN_DOCCLOSED	unDocClosed	VnDocClosed(int)	表示文档已经被关闭
EV_VN_COMMIT	unCommit	VnCommit(BOOL)	表示执行对文档数据的修改
EV_VN_REVERT	vnRevert	VnRevert(BOOL)	表示放弃对数据的修改,从文档中恢复数据
EV_VN_ISDIRTY	vnIsDirty	VnIsDirty(void)	如果 view 中数据被修改过且还没有执行这些修改,则返回 FALSE
EV_VN_ISVINDOW	vnIsWindow	VnIsWIndow(HWND)	如果 HWND 是 Views 的显示窗口返回 TRUE

所有这些消息的处理函数均返回 BOOL。

(二)增加用户的浏览器事件

使用 VN_DEFINE 和 NOTIFY_SIG 宏定义来处理用户自己的浏览器事件,定义相应的响应表宏定义和事件处理函数。这一部分讨论如何定义一个事件,建立事件处理函数以及事件的响应表宏定义。

首先,必须定义想处理的事件名;一般,事件名以 vn 开头,后跟事件名。用户的浏览器事件应该定义成一个 const int,其值要大于 vnCustomBase,可以将事件值定义为 vnCustomBase 加上某个偏移值。例如:定义一个 vnPenChange 事件0其定义可能如下:

```
const int vnPenChange=vnCustomBase+1;
```

下一步,用 NOTIFY_SIG 宏指定事件处理函数。该宏带两个参数;第一个参数是事件名,第二个参数是真正要传给函数的参数类型。这个参数的大小不会比 long 型大;如果被传递的对象大于 long,必须用指针传递。例如:假设对 vnPenChange 事件,想传一个 TPen 对象给事件处理函数。由于 TPen 对象比“long”大得多,必须通过指针传递对象,宏定义如下:

```
NOTIFY_SIG(vnPenChange,TPen *);
```

现在,需要为事件定义事件响应表宏。一般宏名用事件名,全部大写,以 EV_VN 开关,用#define 定义宏名,用 VN_DEFINE 宏定义宏本身,这个宏带三个参数:

- ■ 宏名
- ■ 事件处理函数名(约定同事件名,但以 Vn 而不是 vn 开头)。
- ■ 事件处理函数的大小。有四种可能的值:
  - · void

- int（整型参数，依赖于不同的平台）
- long（32位整数或指针）
- 指针（指针大小，依赖于内存模式）

vnPenChange 事件的响应表宏定义如下：

```
#define EV_VN_PENCHANGE\
VN_DEFINE(vnPenChange,vnPenChange,pointer)
```

注意：*VN_DEFINE* 宏的第三个参数是一个指针，它指明传给事件处理函数的值的大小。

2.10.7 Doc/View **属性**

每一个文档和浏览器对象均有一个属性链表，还有一系列函数，通过这些函数查询和改变这些属性。属性包含有对象及能力的有关信息。当文档管理器创建或撤除文档和浏览器对象时，它便向应用程序发一个通知消息，应用程序查询对象的属性以决定如何处理。浏览器也可以存取相关文档的属性。

一、属性名属性值

TDocument 和 TView 有一些通用属性，从 TDocument 和 TView 派生的类都可获取这些属性。属性是由一个 enum 类来索引的。TDocument 和 TView 派生类的第一个属性应该是 PrevProperty，最后一个属性是 NextProperty，这两个值是一每一个文档和浏览器对象的属性界限，它们保证属性值以正确的值开始，不会超过另一个属性值，并允许派生类的属性从一个合适的值开始。

PrevProperty 应该设为“最派生基类”的 NextPoperty-1。

例如：TDocument 派生类的属性表如下：

```
enum{
    prevProperty=TDocument::NextProperty-1,
    Size,
    StorageSize,
    NextProperty,
};
```

注意：在没 *PrevProperty* 时要用范围操作符(::)，这可以保证将 *PrevProperty* 设为合适的值。

属性名一般包含在一个字符串数组中，每个名在数据中的位置和枚举属性索引相对应。但在派生类中增加索引时，可以用任何方式来存取字符串。由于用户必须写函数来存取属性，不推荐用很复杂的存储方法。属性名应该是属性的简单描述。

属性的属性值同样包含在一个数组中，定次是一个 int 数据获取数组中属性的属性值

的一般做法是用相对应的,属性索引的值。属性值表示如何操作属性。

GetText	以文本格式获取属性
pfGetBinary	以非文本格式获取属性
pfconstant	一旦创建了对象,就不能更改属性
pfSettable	属性可以设置,必须提供原始格式
pfUnknown	对象中定义了属性,但不能获取
pfHidden	与一般浏览时,属性要被隐藏(不准用户看到名字或值)
pfUserDef	属性值在运行时由用户指定

二、获取属性信息

TDocument 和 TView 另提供了许多函数来操作 Doc/View 对象的属性信息,所有这些函数均为 virtual。由于属性操作函数是 virtual 函数,所以绝大多数派生类中的函数可以重载定义在基类中的属性。每个类要负责完成属性的存取。

通常由索引值来操作属性。使用带属性名的 FindProperty 函数,该函数带一个 char * 参数,在属性链表中搜索同名的属性,函数返回一个 int 值,它是属性的索引值,供后续调用使用。

也可用 PropertyName 函数来查找属性名,该函数带一个 int 参数,返回包含属性名的 char * 指针。

用 PropertyFlags 函数获取属性的属性值,该函数带一个 int 参数,即欲查找的索引,函数返回 int,用与操作(&)来检查某个标志是否被设置。如:检测文本形式下的属性值,代码如下:

```
if(doc-> PropertyFlags() &pfGetText){
    // Get Property as text...
}
```

在上面例子中,我们检测 pfGetText 标志。

三、获得和设置属性

用 GetProperty 和 SetProperty 函数来查询和修改 Doc/View 对象的。

GetProperty 函数用来查找属性值:

```
int GetProperty(int index,void far * dest,int textlen=0);
```

这里:

- index 是属性的索引值
- dest 用来包含属性数据;
- textlen 是 dest 指针指向的内存数组的大小。如果 textles 为0,则属性数据以二进制方式返回,否则以文本方式返回,只有 pfGetBinary 属性值设置之后,数据才能以二进制方式返回,同样,只有当 pfGetText 标志设定以后,数据才能以文本方式返回,要获取或设置属性的二进制数据,调用者必须知道数据的类型和语义。

SetProperty 函数用来设置属性值：

```
BOOL SetProperty(int index,const vid far * src);
```

这里：

- index 是属性索引
- src 是欲设置属性的数据，src 必须以原始格式设置；一个派生类要复制属性名时，也应该提供相同的行为如数据类型。

2.11 合法性检验器对象

ObjectWindows 提供了几种途径，用来联系合法性检验器和编辑控制对象，可以检验用户在编辑控制中的信息是否合法。运用合法性检验器对象可以很容易地在 ObjectWindows 应用程序中增加合法性数据检验或者改变数据合法性检验方法。

- 使用标准的合法性检验器类。
- 使用数据合法性检验器对象。
- 编写自己的合法性检验器对象。

任何时候都可以调用对象的成员函数 CanClose 检验编辑控制中内容的合法性，这个函数顺序调用适当的合法性检验器对象。合法性检验器还可以在击键和聚焦/失去焦点的层次上交互作用。

2.11.1 标准的合法性检验器类

ObjectWindows 的标准合法性检验器类自动地进行数据合法性检验，ObjectWindows 在头文件 validate.h 中定义了六种合法性检验器类。

- 类 TValidator 是一个基类，可以派生出所有其他的合法性检验器类。
- 类 TFilterValidator，是一个过滤式合法性检验器类。
- 类 TRangeValidator，是一个数字范围的合法性检验器类，它是类 TFilterValidator 的派生类。
- 类 TLookupValidator，是一个查询合法性检验器基类。
- 类 TStringLookupValidator，是一个字符串查询合法性检验器类，它是类 TLookup-Validator 的派生类。
- 类 TPXPictureValidator，是一个图像合法性检验器类，用于检验给定模式下字符串或“图像”的合法性。

下面的内容简单地描述了每个标准的合法性检验器类。

一、合法性检验器基类

抽象类 TValidator 是所有合法性检验器类的基类，类 TValidator 是一个合法性检验器，对于所有输入都是合法的：成员函数 IsValid 和 IsValidInput 总是返回真值 TRUE，而函数 Error 则不起任何作用。派生类应该重载函数 IsValid，IsValidInput 和 Error，定义什么样的值

是合法的，什么时候应该报告出错。如果没有其他合适的合法性检验器类作为起始点，则可以用类 TValidator 作为自己的合法性检验器类的起始点。

二、过滤式合法性检验器类

类 TFilterValidator 是一个简单的合法性检验器，用来检验用户输入的合法性，这个过滤合法性检验器的构造函数只有一个参数，是一个有效的字符串集：

```
TFilterValidator(const TCharSet& validchars);
```

这里 TCharSet 在头文件 bitset.h 中有定义。

类 TFilterValidator 重载函数 IsValidInput，只有当前输入的字符串中的字符都包含在传给构造函数的字符串集中的字符时，才返回真值。只有当函数 IsValidInput 返回 TRUE 时，编辑控制才插入字符，所以没有必要重载函数 IsValid，因为这些字符都通过了输入过滤器，得到完整的字符串肯定是有效的，类 TFilterValidator 的派生类，像 TRangeValidator，可以对输入后的字符串进行过滤和其他方法综合检查。

三、范围合法性检验器类

类 TRangeValidator 是类 TFilterValidator 的派生类，它进行一定范围的合法性检验，它只接受数据，并在最后结果中增加范围检查，其构造函数有两个参数：分别定义最大和最小的有效值：

```
TRangeValidator(long min,log mar);
```

范围合法性检验器在构造自己时，像过滤式合法性检验器那样，只接受数字0到9和正、负号，因此，继承的函数 IsValidInput 保证只有数字滤过。类 TRangeValidator 重载函数 IsValid，只有当输入的数字是一个有效的整数，而且是在构造函数定义的范围之间时，才返回 TRUE，成员函数 Error 显示一个信息框，表示输入的值超出范围。

四、查询式合法性检验器类

类 TLookupValidator 是一个抽象基本类，它将输入的值与一系列可接受的值进行比较，决定其是否有效。类定义虚拟成员函数 Lookup，默认状态下，函数 Lookup 返回真值 TRUE。派生类应当重载函数 Lookup，将参数与各项进行比较，如何匹配成功，则返回真值。

TLookupValidator 重载函数 IsValid，仅当函数 Lookup 返回 TRUE 时它才返回 TRUE，在派生类中应当重载函数 IsValid，而不必重载函数 Lookup。

类 TStringLooupValidator 是基于类 TLookupValidator 的一个事例类。

五、字符串查询合法性检验类

类 TStringLookupValidator 是一个查询合法性检验器的工作实例；它将传给编辑控制的字符串与字符串表中的各项进行比较，如果传过来的字符串出现在字符串表中，函数 IsValid 返回真值 TRUE，这个构造函数只有一个参数，是有效字符串的列表：

```
TStringLookupValidator(TSortedStringArray * strings);
```

TSortedStringArray 是这样定义的：

在构造了字符串查询合法性检验器后，为了使用一个不同的字符串表，使用成员函数 NewStringList，它除去了字符串表并安装新的字符串表。

类 TstringLookupValidator 重载函数 Lookup 和 Error。当传入的字符串是字符串表中之一时，函数 Lookup 返回真值 TRUE，如果不是则函数 Error 显示一个消息框。

六、图像合法性检验器

图像合法性检验器将用户输入的字符串与一个"图像"或描述合法输入格式的样板进行比较。这里用到的图像与 Borlands'Paradox 的用于控制用户输入的图像是兼容的。构造一个图像合法性检验器需要两个参数：一个存储样板映像的串，和一个显示是否用合法字符填入图像中的 Boolean 型值：

```
TPXPictureValidator(const Char for * pil,BOOL autoFill=FALSE);
```

类 TPXPictureValidator 重载函数 Error，IsValid 和 IsValidInput，并增加了一个成员函数 Picture。函数 Error 显示一个消息框表示字符串格式应该是什么样子。函数 IsValid 只有在函数 Picture 返回 TRUE 时返回 TRUE；因此，只要重载成员函数 Picture，就可以派生出一个新的图像合法性检验器。函数 IsValidInput 在用户输入时检测字符，只允许图像格式承认的字符，有选择地用图像格式中的合法字符进行填充。

下面是一个图像合法性检验器的实例，用于接受社会安全号码：

```
edit -> SetValidator(new TPXPictureValidator
("=========="));
```

图像语法在《ObjectWindows Reference Guide》一书中关于类 TPXPictureValidator 成员函数 Pciture 部分有详细描述。

Picture 成员函数力图根据图像格式格式化给定的输入串，并返回显示其成功程度的值，下面的代码列出了这些返回的值：

```
//
// TPXPictureValidator result type
//
enum    TPicResult {
        prComplete,
        prIncomplete,
        prEmpty,
        prError,
        prSyntax,
        prAmbiguous,
        prIncompNoFill
};
```

2.11.2　使用数据合法性检验器

为了使用数据合法性检验器对象，必须构造一个编辑控制对象，并构造一个合法性检验

器对象同时将其定义给编辑控制。从这一点看，设计者不必直接同合法性检验器对象交互打交道，编辑控制知道在适当的时候调用合法性检验器成员函数。

一、构造一个编辑控制对象

编辑控制对象是类 TEdit 的实例，下面是构造一个编辑控制的例子：

```
TEdit * edit;
edit=new TEdit (this,101,sizeof (transfer,NameEdit));
```

更多的关于 TEdit 和使用编辑控制的信息参见第五节。

二、构造并分配合法性检验器对象

因为合法性检验器对象不是界面对象，它们的构造函数只需要足够的消息建立合法性检验准则即可。例如，一个数字范围的合法性检验器对象只要求两个参数：表示有效范围的最小值和最大值。

每一个编辑控制对象都有一个数据成员，可以指向一个合法性检验器对象，指针的声明如下：

```
TValidator * Validator
```

如果 Validator 不指向一个合法性检验器对象，则编辑控制行为像第五章描述的那样，可以通过调用编辑控对象的成员函数 SetValidator，确定一个合法性检验器。编辑控制在处理关键事件或调用检验自身全法性时，自动用合法性检验器进行检查。

下面的代码构造了一个合法性检验器并指定了一个编辑控制，在这里，用到了一个只允许字母字符的过滤合法性检验器。

```
edit->SetValidator(new TFilterValidator("A-Za-Z"));
```

在 OWLAPI\VALIDATE 中，有用到标准合法性检验器的完整例子。

2.11.3 重载合法性检验器的成员函数

虽然标准的合法性检验器对象能满足大多数的数据合法性检验需要，但是还可以修改标准的合法性检验器或重新定义自己的合法性检验器对象，如果准备这么样，就必须熟悉下列从基类 TValidator 继承下来的成员函数，除了理解每个成员函数的功能外，还必须知道编辑控制如何利用它们，以及需要怎样重载这些函数。

- Valid
- IsValid
- IsValidInput
- Error

一、成员函数 Valid

成员函数 Valid 是由相关的编辑控制对象调用的，它用于验证输入的数据是否合法，很像界面对象中的成员函数 CanClose，Valid 是一个 BOOL 型函数，只有当传过来的串是合法

的情况下才返回 TRUE。编辑控制的成员函数 CanClose 的其中一个功用就是调用合法性检验器的成员函数 Valid，传递编辑控制器的当前文本。

在编辑控制中使用合法性检验器时，不必调用或重载合法性检验器的成员函数 Valid；从父类继承下来的函数 Valid 版本就足够了。默认状态下，如果成员函数 IsValid 返回 TRUE，则 Valid 也返回 TRUE；否则，调用函数 Error 通知用户出错，并返回 FALSE。

二、成员函数 IsValid

函数 Valid 调用虚拟成员函数 IsValid，它传递给函数 IsValid 需要检验的文本串，如果这个串代表着有效的数据，则函数 IsValid 返回 TRUE。

函数 IsValid 完成具体的数据合法性检验过程，所以如果需创建自己的合法性检验对象时，就可能要重载函数 IsValid。

注意：不要直接调用函数 *IsValid*，应该用函数 *Valid* 调用 *IsValid*，因为如果函数 *IsValid* 返回 *FALSE* 时，函数需要调用 *Error* 警告用户出错信息，这样，就将合法性检验工作与出错报告分离开来。

三、成员函数 IsValidInput

当一个编辑控制对象发现其中有击键事件发生时，它调用其合法性检验器的成员函数，以保证键入的字符是个有效的输入。默认状态下，成员函数 IsValidInput 通常返回 TRUE，意谓着所有的击键都是可接受的，但是，一些派生的合法性检验器重载函数 IsValidInput 滤去不想要的击键。

例如，合法范围的检验器，用于输入数据的合法性检验，只有当输入全为'+'、'-'或数字时，函数 IsValidInput 才返回 TRUE。

函数 IsValidInput 有两个参数：

```
virtual BOOL IsValidInput(Char far * str,BOOL suppressFile);
```

第一个参数 str 指向当前输入的正在进行合法性检验的文本，第二个参数是个 Boolean 类型值，显示在准备进行合法性检验之前，是否要填充或增补输入的串。TPXPictureValidator 是使用第二个参数的唯一的标准合法性检验器对象。

四、成员函数 Error

虚拟成员函数 Error 警告用户，编辑控制中的内容没有通过合法性检验器的检查，标准的合法性检验器对象通常提供一个简单的消息框，通知用户输入的内容是无效的，并描述什么样的输入才是有效的。

例如，一个进行范围检查的合法性检验器对象的成员函数 Error 创建一个消息框，显示一编辑控制中的值不在提供的最小值和最大值之间。

虽然大多数的合法性检验器的子孙对象都重载函数 Error，但不必直接调用它，如果函数 IsValid 返回 FALSE，函数 Valid 会自动调用 Error，这是函数 Error 唯一需要调用的地方。

2.12 打印机对象

本章介绍如何利用 ObjectWindows 完成下面几项打印任务：

- 创建一个打印机对象
- 创建一个打印机输出对象
- 打印窗口内容
- 打印文档
- 选择、配置打印机

如何应用 TPrinter 和 TPrintout 类，完成上述五项任务将是很轻松的事情：

- TPrinter 封装了打印机的功能及其及使用打印机驱动程序的方法。它显示一个对话框，在该对话框内用户选择和设置打印机。
- TPrintout 封装了实际的打印输出动作。它与打印机之间的关系同 TWindow 与屏幕之间的关系很相似。TWindow 对象的成员函数 Paint 在屏幕上绘制图形，TPrintout 对象的成员函数 PrintPage 在打印机上打印信息。为了实现打印机输出，需要为 TPrinter 的 print 成员函数传递一个 TPrinter 实例。

2.12.1 创建打印机对象

创建打印机对象的最简单方法是在窗口对象中声明一个指向 TPrinter 对象的指针，即 TPrinter *，以备其他对象打印时使用：

```
class MyWindow:public TFrameWindow{
    TPrinter * printer;
    // ...
};
```

为了使用指针 Printer，需要在构造函数 MyWindow 中使其指向一个 TPrinter 类的实例：

```
MyWindow::MyWindow(TWindow * parent,char * title){
    // ...
    Printer=new TPrinter;
}
```

在 MyWindow 类的析构函数中删除 Printer 对象：

```
MyWindow::~MyWindow(){
    // ...
    delete Printer;
}
```

下面的程序定义一个打印机窗口类，在该类中定义一个指向 TPrinter 指针，在构造函数中，初始化指向打印机的指针，程序如下：

```
classTprinterWindow:public TFrameWindow {
{
    TPrinter * Printer;
    //...
};
    TPrinterWindow::TPrinterWindow(TWindow * Parent,const char * litle,
                                   TModule * module)
        :TFrameWindow(parent,title,0,FALSE,module),
        TWindow(Parent,title,module)
{
    //...
    Printer=new TPrinter;
}
```

对于大多数应用程序，上面介绍的方法已经足够了。即在应用程序的主窗口初始化打印机对象，使用 WIN.INI 定义的缺省的打印机。然而在各别的应用程序中，会有几个不同的窗口同时应用不同类型的打印机的情况，在这种情况下，需要在每个窗口的构造函数中构造一个打印机对象，然后根据需要每个打印机对象改变打印设备。如果是在不同的时间使用不同的打印机，最好是使用一个打印机对象，根据实际需要选用不同的打印设备。

如果为了使用非缺省的打印机而而欲重载 TPrinter 的构造函数。建议使用缺省的构造函数，然后改变与打印机对象相关联的设备，请参照本章第五节。

2.12.2 创建打印输出对象

创建一个打印输出对象与为窗口对象编写一个 Paint 函数十分类似：Paint 函数利用 Windows 提供的图形函数在设备上下文上绘图，由窗口对象的显示下下文管理与屏幕之间的交互作用：与此类似打印输出对象的设备上下文封装了打印机的功能及使用方法，避免了程序员直接与打印机打交道。

为了创建打印输出对象：

- 以 TPrinter 为基类派生一个新类，在该新类中重载成员函数 PrintPage，如果打印的页数仅有一页，没有必要进行下一步。
- 如果打印的文档具有多页，还需要重载成员函数 HasPage，当还有欲打印的页时，该函数必须返回一个非零值。当前页的顺序号以参数的形式传递给 PrintPage 函数。

打印输出对象有用来保存页的尺寸及已初始化的设备上下文的数据成员。打印机对象通过调用打印输出对象的 SetPrintParams 函数把这些成员的值发送给打印机。在每次调用 Windows 提供的图形函数时都要用到打印输出对象的设备上下文。

下面程序代码定义一个 TPrinter 的派生类，在该类中，重载了基类 TPrinter 的一些函数。

```
class TWindowPrintout::public TPrintout{
  public:
    TWindowPrintout(const char * title,TWindow * window);
```

```
    void GetDialogInfo(int& minPage,int& maxPage,int& selFram Page,int& SelToPage);
    void PrintPage(int page,Trect& rect,unsigned flags);
    void SetBanding(BOOL 6){Banding=b;]
    BOOL HasPage( int pageNumber) {reutrn pageNumber==1;}
    protected:
      TWindow * Window;
      BOOL        scale;
};
```

如果设置了页面选择功能,GetDialogInfo 将出弹出了一个对话框提示用户输入页面尺寸,然后将其返回。由于仅有一页,GetDialogInfo 函数的代码为:

```
TWindowPrintout::GetDialogInfo(int& minPage,int& maxPage,
                                          int& selFrompage,int& selToPage)
{
    minPage=0;
    maxPage=0;
    selFromePage=0;
    SelToPage=0;
}
```

为了打印各页的内容、装订线(如果设置了装订线功能)、窗口,需要重载 PrintPage 函数,其代码如下:

```
void TWindowPrintout::PrintPage(int,TRect& rect,unsigned)
{
    // conditionally scale the DC to the Window so
    // the printout will resemble the window
    int prevMode
    TSize oldVExt,oldWExt;
    if(Scale){
      prevMode=DC->SetMapMode(MM_ISOTROPIC);
      TRect WindowSize=Window->GetClientRect();
      DC->SetViewPortExt(pageSize,&oldVExt);
      DC->SetWindowExt(windowSize.Size(),&oldWExt);
      DC->IntersectclipRect(windowSize);
      DC->DPtoLP(rect,2);
        }
    // call the window to Paint itself
      Window->Paint( * DC,FALSE,rect);
    // Restore change made to the DC
      if(Scale){
        DC->SetWindowExt(odlWext);
        DC->SetViewPortExt(oldVExt);
        DC->SetMapMode(prevMode);
```

```
    }
}
```

欲设置装订线功能，则调用 SetBanding 函数：

```
printout. SetBanding(TRUE);
```

每次打印完一页纸，都调用 HasPage 函数检查是否尚有未打印的页。在缺省条件下，该函数返回 FALSE，意味所打印的文档仅有一页。如果打印的是一个多页文档，必须重载 HasPage 函数，在有未打印页的情况下返回 TRUE。

2.12.3 打印窗口内容

最简的打印输出是拷贝窗口内容，因为它相当于一个单页的输出，而且窗口对象知道如何在设备上下文中绘制自己。

创建一个窗口打印输出对象的方法是：先构造一个窗口打印输出对象，然后传递给它两个参数：一个是标题字串，一个指欲打印窗口的指针。

```
TWindowPrintout Printout("Ruler Test",this);
```

通常，希望窗口能以响应菜单命令的方式打印自己。下面的程序代码定义了响应窗口打印命令的消息响应函数：

```
void TPrinterWindow:CmFilePrint()    // Execute File:Print command
{
    if(Printer){
        TWindowPrintout printout("Ruler Test",this);
        Printout. SetBanding(TRUE);
        printer->Print(this,printout,TRUE);
    }
}
```

该函数调用了打印机对象的成员函数 Print。Print 函数传递一个指向父窗口的指针，一个指向打印输出对象的指针，以及指示是否显示打印机对话框的标志量。

TWindowPrintout 通过调用窗口对象的 Print 函数实现自身的打印(在 TWindowPrintout::PrintPage)这里它使用的是打印机设备上下文而不显示上下文。

2.12.4 打印文档

Windows 将一个打印输出识为若干欲打印的页。因此打印输出对象必须把欲打印的文档分解为若干页映像。如同利用窗口对象在屏幕上为 Windows 绘制图形一样，打印机输出对象在打印机上绘制图形。

打印输出对象需要完成下面几项工作：

- 设置打印参数
- 计算打印的总页数
- 在设备上下文中绘制每一页

■ 显示是否还有更多的页

一、设置打印参数

为了文档能自动标定自己的页数，打印机对象(由 TPrinter)调用两个打印输出对象的成员函数:SetPrintParams 和 GetDialogInfo。

SetPrintParams 初始化页尺寸及设备上下文变量，同时计算打印单独的某一页时所需要的信息量。例如，SetPrintParams 可以计算出以选定的字体为基准，在打印区域内包含了多少文本(应用 Windows API GetTextMetrics)。如果重载 SetPrintParams 函数，一定要调用基类的 SetPrintParams 函数来设置缺省的页尺寸及设备上下文。

二、计算页数

调用 SetPrintParams 之后，打印机对象调用 GetDialogInfo 函数，该函数通过打印机对话框返回用户的页尺寸信息。该尺寸被用来计算打印的总页数。

三、打印每一页

在打印机对话框为打印的文档提供标写自己页数的机会后，调用打印输出对象的 PrintPage 函数打印每一页。打印指定页上的部分文档与确定在滚动窗口内的那一部分需要绘制是一致的。

在编写 PrintPage 函数时，需要记住下面两个原则:

- 设备无关性　不要对比例、尺寸、颜色等硬编码。因为这些信息随显示卡及打印设备的不同而不同，所以应去掉任何依赖于具体设备的代码。
- 设备硬件性能　尽管大部分视频设备都支持所有的 GDI 函数，但有些打印机不支持。例如许多打印外设，比方绘图仪，根本不接受 Windows 位图(bitmap)。个别的外设支持部分 GDI 函数，当执行一项复杂的输出任务时，应调用 Windows API 函数 GetDeviceCaps，该函数可以返回给定外设的一些重要性能参数。

四、指示是否有待打印的页

打印输出对象的最后一项职责:通知打印机对象在给定的页之后是否还有可打印的页。HasPage 函数带一个参数，即一个页数，返回一个布尔值，该返回值指出是否还有没打印的页存在。在缺省情况下，该函数仅对于第一页返回 TRUE，为了打印多页文档，需要重载 HasPage 函数，使之在存在待打印页时返回值 TRUE，而对于最后一页返回 FALSE。

一定确保 HasPage 函数在适当的时候返回 FALSE。如果该函数总返回 TRUE，打印输入将会陷于死循环。

五、有关打印输出的其他考虑

打印输出对象还有几个其他成员函数在需要时可以重载。例如，BeginPrinting 和 EndPrinting 在打印文档前、后要分别调用。如果需要特殊的启动代码，可以把它加到 BeginPrinting 函数中，在 EndPrinting 函数中撤销它。

文档页的打印是按顺序进行的。即打印机对象调用 PrintPage 函数打印每一页。在第一

次调用 PrintPage 之前，打印机对象先调用 BeginDocument，并给它传递两个参数，文档打印的第一页和最后一页。如果文档不是从其第一页开始打印而做预备工作时，需要重载 BeginDocument 函数。相对应的函数为 EndDocument。在打印完最后一页时，需要调用 EndDocument。

如果文档欲打印多份，在 BeginPrinting 和 EndPrinting 之间要多次调用 BeginDocument/EndDocument。

五、选择不同的打印机

在应用程序当中，可以将打印机对象与任何一种打印机设备相关联。在缺省条件下，TPrinter 应用 Windows 在 WIN. INI 文件中定义的缺省的打印机。

有两种方法确定选用一个不同打印机：直接（在编码时）和应用对话框。

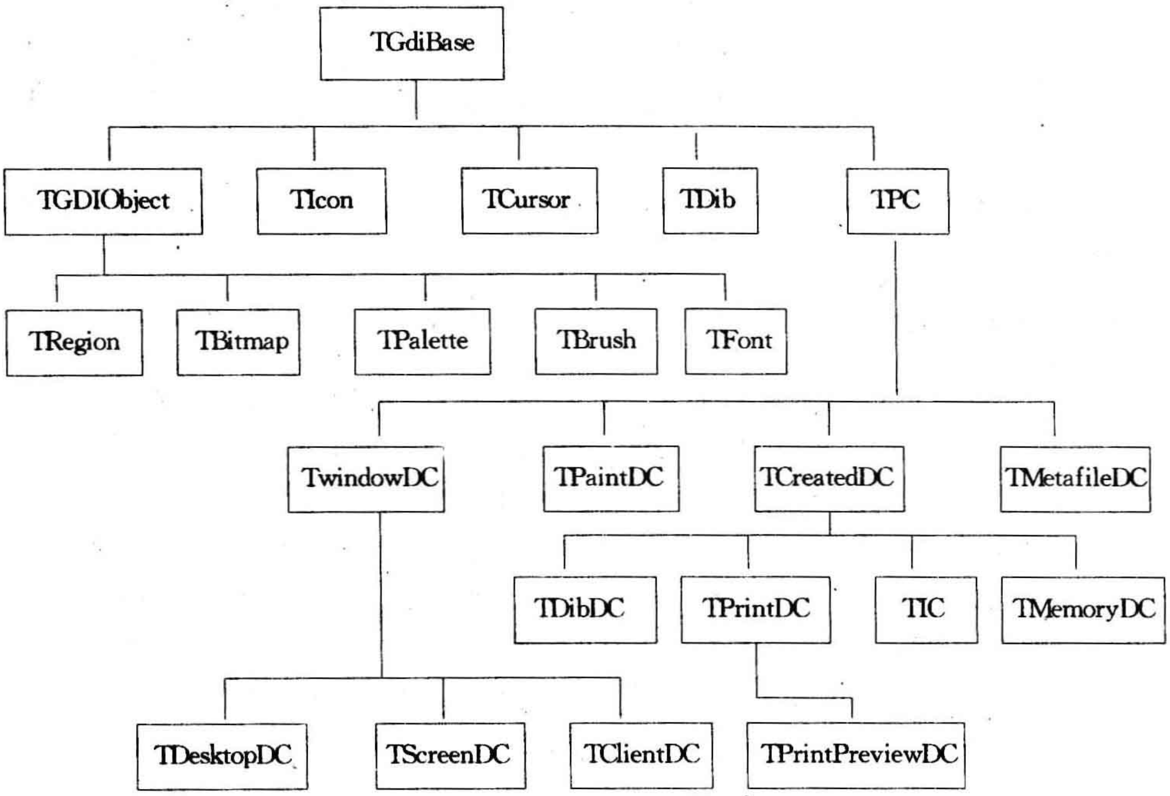
至今为止，选择一个不同打印机的最通用的方法是弹出一个对话框。在该对话框内列举出所有已安装的打印外设。当调用 TPrinter 的成员函数 Setup 时，TPrinter 自动完成这些任务。Setup 函数根据 TPrinterDialog 类弹出一个对话框。

在对话框中有一个 Setup 按钮，当用户按下该按钮时可以对打印机进行配制。Setup 按钮弹出一个打印机配制对话框引导用户完成此项工作。用户程序无法控制该对话框的外观及功能。

在有些情况下，调用 SetPrinter 函数可以直接为打印机对象设定一个打印机设备，而不需要用户输入。SetPrint 带三个字串参数：一个设备名字、一个驱动程序名、一个端口名字。

2.13 总 结

TPrintert TPrintout 类封装了对打印机的操作，利用这两个类提供的接口函数，程序员可以方便的操作打印机。在 Broland C++ 3.1中要方便地做到这些功能，程序员的确要费一番心思，相信编写过类似程序的读者会有同感。总结：BorlandC++ 4.0 OWL 2.0将图形设备接口部分操作封装为一个类体系。该类体系结构参见联机帮助！



在这一部分,我们只介绍了其中一部分内容。

第三章 ObjectWindows 动态链接库

动态链接库(DLL)是一个含有函数、数据和资源的库,它们是在运行时引用而不是编译时引用。使用静态链接库的应用程序在编辑时即链入库中的那段代码,而使用 DLLs 的应用程序与其他使用 DLL 的应用程序共享那一段代码,因此,应用程序变得更短小精悍。例如,如果想定义复杂的窗口行为,在一个 ObjectWindows DLL 中,一组应用程序可以共享这段代码。

这一章描述如何编写和使用 ObjectWindows DLLs。

3.1 编写 DLL 函数

编写应用程序中调用的 DLL 函数时,应注意以下几件事:

- 调用16位的 DLL 函数时,应该是远程调用,同样,标明参数和返回值的指针也应该是远程指针,之所以这么做,是因为16位的 DLL 与调用应用程序有不同的代码和数据段(32位的 DLLs 就不必这样),用宏_FAR 使得代码能在各种支撑平台之间兼容。
- 在16位 DLL 中定义的静态数据对于所有调用的应用程序是全局型的,因为16位的 DLLs 有一数据段可以由所有16位的 DLL 事例共享,某一应用程序设置的数据可以被另外的应用程序访问。如果希望某个数据是私有的,只被一个16位的 DLL 应用程序调用,则必须在单个任务中动态地分配和管理这个数据,对于32位的 DLLs,静态数据对于每个过程是私有的。

3.1.1 DLL 的入口和出口函数

Windows 在每个 DLL 中定义两个函数:一个入口函数和一个出口函数。对于16位的 DLLs,入口函数由 LibMain 调用,而出口函数由 WEP(Windows Exit Procedure)调用,Windows 调用函数 LibMain 是应用程序调用 DLL 的开始,而 Windows 调用函数 WEP 则意味着应用程序调用 DLL 的结束。对于32位的 DLLs,函数 DLLEntryPoint 既作为入口函数又作为出口函数。每次加载和卸载 DLL 时都调用函数 DLLEntryPoint,相应地每次都有一个 DLL 中的过程与应用程序相链接和分离,并且伴随着过程内的联系创建或消除。

当 DLL 库第一次被启动时,Windows 调用入口过程(LibMain 或者 DllEntryPoint)一次,入口程序初始化 DLL,这个初始化工作几乎依赖于特定的 DLL 函数,但可能包含下列任务:

- 如果 UnlockData 声明为 MOVEABLE,则用它解锁数据段。
- 如果需要,为 DLL 设置全局变量。

由于 DLL 初始化代码(CoDx.OBJ)自动初始化局部堆,所以不必注意堆的初始化工作,

下面的内容描述了16位和32位的 DLL 入口和出口函数。

3.1.2 LbiMain 过程

16位 DLL 的入口过程 LibMain 定义如下:

```
int FAR PASCAL LibMain(HINSTANCE hinstance,WORD wDataSeg,
                       WORD cbHeapSize,LPSTR lpCmdLine);
```

LibMain,HINSTANCE,WORD 和 LPSTR 在头文件 Window.h 中定义。

参数描述如下:

- hInstance 是 DLL 的事件句柄。
- wDataSeg 是数据段(DS)寄存器的值。

在入口函数内,TModule 对象 ResMod 用 DLL 的事例句柄初始化。如果这个模块没有装入,则有例外。

如果 DLL 需要另外的初始化和清理工作,那么可以在自己的 LibMain、DllEntryPoint 或 WEP 函数中执行这个过程,一个最好的办法是派生一个 TModule 类,在类中为 DLL 定义全局的数据成员,在构造函数和析构函数中执行要求的初始化和清理工作。

3.1.3 作为一个 DLL 使用 ObjectWindows

为了使 ObjectWindows 应用程序能共享 ObjectWindows 库的简单拷贝,必须将它们动态地链入 ObjectWindows DLL。为了这么做,必须保证以下几件事:

- 在编译时,在命令行或在 IDE 中增加宏_OWLDLL 的定义。
- 链接时,不指明静态 ObjectWindows 链接库(即:OWLWS.LIB,OWLWM.LIB,OWLWL.LIB 或 OWLWF.LIB),而指明 ObjectWindows DLL 输入库(16位应用程序是 OWLWI.LIB,32位应用程序是 OWLWFI.LIB)。
- cbHeapSize 是 DLL 的模式定义文件中指定的局部堆的大小。
- lpCmdLine 是启动 DLL 时,确定的命令行的远程指针,它通常是空的,因为一般情况下,DLL 调用自动完成,不需要任何参数,但是,如果需要的话,当 DLL 显示调用时,提供一个命令行作为参数。

函数 LibMain 的返回值或者是1(初始化成功)或者为0(初始化失败),如果返回值是0,Windows 从内存中释放 DLL。

3.1.4 WEP 过程

函数 WEP 是 DLL 的出口过程,Windows 在释放 DLL 之前调用它,这个函数在 DLL 中不是必需的(因为 Borland C++运行库提供了一个默认的函数),但是可以通过 DLL 编写器提供实用的程序,在 DLL 从内存中释放之前完成各种清理工作。通常在调用 WEP 时,应用程序结束工作,所以有效的操作至此结束。

在 Borland C++环境下,函数 WEP 不需出口,这里是函数 WEP 的原型:

```
int FAR PASCAL WEP(int nParameter);
```

参数 nParameter 或者是 WEP_SYSTEMEXIT，表示窗口正在关闭；或者是 WEP_FREE_DLL，表示正在释放 DLL。WEP 返回值为1表示成功。Windows 现在不使用这个返回值。

3.1.5 DLLEntryPoint 函数

32位 DLL 的入口点函数 DLLEntryPoint 定义如下：

```
BOOL WINAPI DLLEntryPoint(HINSTANCE hinstDLL,
                          DWORD fdwReason,LPVOID lpvReserved)
```

DLLEntryPoint 在头文件 Window.h 中定义。

参数描述如下：

- histDLL 是 DLL 事件句柄
- fdwReason 是一个标志，描述调用 DLL 的原因(一个过程或一个动作)，这个标志可以取下列值：
  - DLL_PROCESS_ATTACH
  - DLL_THREAD_ATTACH
  - DLL_THREAD_DETACH
  - DLL_PROCESS_DETACH
- LpvReserved 进一步指明 DLL 初始化和清理工作的各个方面，它基于值 fdwReason。

3.1.6 输出 DLL 函数

编写完自己的 DLL 函数，必须输出这个函数，以便能被应用程调用，包含两步：将自己编写的 DLL 函数作为可输出函数编译，并输出它们。

可以在下面几步中完成这个工作：

- 如果用关键字_export 标志一个函数，则按可输出编译并输出。
- 如果在类声明中增加关键字_export，则整个类(包含数据和成员函数)按可输出编译并输出。
- 如果不用_export 标志一个函数，用合适的编译器切换或 IDE 设置将函数编译成可输出的，然后在模式定义(.DEF)文件 EXPORTS 部分中列出这个文件。

3.1.7 输入(调用)DLL 函数

从应用程序中调用一个 DLL 函数，就像调用应用程序自身定义的函数一样，但必需输入应用程序调用的 DLL 函数。

为了输入一个 DLL 函数，可以做到以下几部分：

- 在调用应用程序的模式定义(.DEF)里，增加 IMPORTS，并把 DLL 函数作为输入。
- 链接一个输入库，其中包含 DLL 函数调用应用程序的输入信息(使用 IMPLB 做输入库)。
- 用函数 LoadLibrary 明确调用 DLL 并用函数 GetProcAddress 获得函数地址。

当应用程序运行时,调用的 DLL 文件必须在当前目录下,如果用路径,则必须在 Windows 中或在 Windows 系统目录下,否则应用程序将找不到 DLL 文件,并且不能加载 DLL。

3.2 编写共享的 ObjectWindows 类

在 DLL 中的类的实例可以由多个应用程序共享,例如,可以在 DLL 中定义一个共享的对话框,使多个应用程序共享这一段对话框的代码。为了共享一个类,必须从 DLL 中输出这个类,并在应用程序中输这个类。

3.2.1 定义共享类

为了定义共享类,必须:

- 有条件地将类声明为_exprot 或_import
- 将一个 TModule * 类型的参数传给窗口的构造函数(在某些情况下)。

如果在头文件里声明了一个共享的类,且这个头文件既在 DLL 中包含,又在使用 DLL 中应用程序中包含,那么在编译 DLL 时应将类声明为_export,而在编译应用程序时将类声明为_Import。为了这么做,可以定义一组宏,有条件的设置其值。当编译 DLL 时设置为_Import。例如:

```
#if defined(BUILDEXAMPLEDLL)
    #define _EXAMPLECLASS _export
#elif defined(USEEXAMPLEDLL)
    #define _EXAMPLECLASS _import
#else
    #define _EXAMPLECLASS
# endif
    Class _EXAMPLECLASS TColorControl:Pulbic Tcontrol {
    Public:
    ...
};
```

通过定义 BUILDEXAMPLEDLL(例如在命令行),在编译 DLL 时,使_EXAMPLECLASS 扩展为_export。这使类输出,并被使用 DLL 的多个应用程序共享。

3.3 TModule 对象

关于类 TModule 的完整描述参见 ObjectWindows 参考指南。

Tmoudle 类的实例像一个 ObjectWindows DLL 的面向对象的界面一样工作。TModule 成员函数提供对窗口、内存管理和错误处理的支持。

下面的例子中的代码显示了一个 TModule 对象的声明和初始化。这个例子是条件编译的,所以无论是16位的(LibMain)或32位的(DLLEntry Point)DLLs 都可以使用同一个源文件。

```
Static Tmodule * ResMod;
#if defined(_WIN32_)
    BOOL WINAPI
    DLLEntryPoint(HINSTANCE instance,DWORD /* flag */,LPVOID)
#else  // defined(_WIN32_)
    int
    FAR PASCAL
    LibMain(HINSTANCE instance,WORD/* wDataseg */,
        WORD   /* cbHeapSize */,Char far * /* cmdLine */)
# end if
    {
        // We're using the DLL and want to use the DLL's resources
        //
        if ( ! ResMod)
            ResMod=new TModule (0,instance);
        return TRUE;
}
```

3.4 从非 ObjectWindows 应用程序中调用 ObjectWindows DLL

当一个子窗口是在 ObjectWindows DLL 中创建的，而其父窗口是在 ObjectWindows 支持父窗口与子窗口的框架通讯，但也可以在非 ObjectWindows 应用程序中准备 DLL。

当一个子窗口是在 ObjectWindows DLL 中创建，而其父窗口是在非 ObjectWindows 应用中创建，则父——子关系必须在 ObjectWindows DLL 中模拟，这是通过在 ObjectWindows DLL 中构造一个别名窗口实现。这个别名窗口与父类窗口相联系，父类窗口的句柄在 DLL 调用中指定。

下面的例子中，输出函数 CreateDLLWindow 在 ObjectWindows DLL 中，这个函数戏可以在 ObjectWindows 中也可以在非 ObjectWindows 应用程序中工作。

```
BOOL far_export
CreateDLLWindow(HWND parentHWnd)
{
TWindow * parentAlias=GetWindowPtr ParntHWnd);
                                        // check if an OWL window
if(!parentALias)
    parentALias=new TWindow(ParentHWnd);  // if not,make an alias
TWindow * window=new TWindow(parentAlias,
                "Hello from a DLL!");
window -> Attr.Style=WS_POPUPWINDOW | WS_CAPTION
                |WS_THICKFRAME|MS+MINIMIZEBOX|WS_MAXIMIZEBOX;
```

```
window->Attr.X=100,Window->Atrr.Y=100;
Window->Attr.W=300;Window->Attr.H=300;
return window-> Create();
}
```

函数 CreateDLLWindow 通过调用 GetWindowPtr,决定是否传递了一个非 ObjectWindows 窗口句柄,如果传过来一个非 ObjectWindows 窗口句柄,则返回0。如果是一个非 ObjectWindows 窗口句柄,则构造一个别名父类窗口对象代替父类窗口工作。

3.5 隐含加载和显式加载

隐含加载是在链接应用程序时用.DEF 文件或输入库完成,当使用 DLL 的应用程序加载时,DLL 由 Windows 加载。

显式加载用于在运行时加载 DLLs,并要求使用 Windows API 函数 Load LLibrary 加载 DLL 和函数返回 DLL 函数地址。

3.6 混合使用静态和动态链接库

ObjectWindows 库是用 BIDS(包容类)库建立的,而后者又是用 C 运行时库建立的。

如果链接 DLL 版本的 ObjectWindows 库,就必须与 DLL 版本的 BIDS 运行时库链接,可以通过定义宏_OWLDLL 做到这一点,并不是可以使用的动态链接库和静态库都能混合,下面是一张表,其中每一行列出了一个可以组合的静态库和动态链接库。

表3.1 允许的库组合

静态库	动态链接库
OWL,BIDS,RTL	(none)
OWL,BIDS,	RTL
OWL	BIDS,RTL
(none)	OWL,BIDS,RTL

第四章 Visual Basic 控制对象

ObjectWindows 可以很容易地在 Windows 应用程序中使用 Visual Basic(VBX)1.0兼容的控制，就像使用标准的 Windows 或 ObjectWindows 控制一样。

VBX 控制提供了一个较宽范围的功能，而标准的 Windows 控制中却没有，有许多 VBX 控制的公共领域和商业软件包可以用于提供。非常漂亮和有用的用户界面。

这一章描述如何设计使用 VBX 控制的应用程序，描述类 TVbxControl 和类 TVbx-EventHandler，解释怎样从 VBX 控制中接受消息，并且显示如何获取和设置一个控制的属性。

4.1 使用 VBX 控制

要在 ObjectWindows 应用程序中，使用 VBX 控制，必须完成以下几步：

■ 在主函数 OWLMain 函数中，调用应用程序对象的 Run 函数之前调用函数 VBXInit 在调用了应用程序对象的 Run 函数之后再调用函数 VBXTerm，函数 VBXTerm 没有参数，函数 OWLMain 可能是这样的：

```
int OWLMain(int arge,char  * argv[])
    { VBXInit(_hInstance);
      return TApplication("Wow!"),Run();
      VBXTerm();
    }
```

这些函数初始化和关闭使用 VBX 控制所需每个实例的宿主环境。

■ 组合基本界面类和 TVbxEventHandler 类派生一个新的类，基本界面类是一个随时可以显示控制的类，如果正在对话框中使用这个控制，就必须与类 TDialog 组合。这段代码如下：

```
Class MyVbxDialog:public TDialog,public TVbxEventerHandler
{ public :
    MyVbxDialog(TWindow * parent,char * name)
       :TDialog(parent,name);
        TWindow(parent,name){}
    DECLARE_RESPONSE_TABLE(MyVbxDialog);
}
```

■ 为父类建立一个响应表，包含所有与控制相关的事件，用宏 EV_VBXEVENTNAMe 为每个控制事件建立一个响应，关于响应表的详细描述见第二章第三节。

■ 创建控制类的父类。可以在创建父类时构造控制，也允许父类自己构造控制，这依赖

于控制是如何使用的。这些内容在后面还有详细讨论。

4.2 VBX 控制类

ObjectWindows 提供两个类用于为 VBX 控制设置界面，这些类是 TVbxControl 和 TVbxEventHandler。

4.2.1 类 TVbxControl

类 TVbxControl 通过完成下面的工作为控制提供了实用的用户界面：

- 构造一个 VBX 控制对象
- 获取和改变控制属性
- 发现控制属性的数目，并转换属性名或从属性索引中反转换。
- 调用 Visual Basic 1.0的标准控制方法 AddItem，Move，Refresh 和 RemoveItem。
- 用类 TVbxControl 的成员函数 GetHCTL 获取控制事件的句柄。

类 TVbxControl 是类 TControl 的派生类，而它是类 TWindow 的派生类，因此 TVbx-Control 非常像其他的基于类 TWindow 的界面元素一样起作用。

4.2.2 TVbxControl 构造函数

类 TVbxControl 有两个构造函数，第一个构造函数通过定义一个 VBX 控制文件名(例如：SWITCH.VBX)、控制 ID、控制类、控制标题、位置和尺寸，来动态地构造一个 VBX 控制。

```
TVbxControl(TWindow * Parent,
            int id,
            const char far * FileName,
            const char far * Class Name,
            const char far * title,
            int x, int y,
            int w, int h,
            TModule * module);
```

其中：

- parent 是一个指向控制父类的指针。
- id 是控制的 ID，它在定义父类的响应表时用到，它看起来很像一个资源 ID。
- FileName 是包含 VBX 控制的文件名，如果需要的话，加上路径名。
- ClassName 是这个控制的类名。一个给定的 VBX 控制文件可能含有许多单独的控制，每个控制由一个单独的类名标识(在第三部分的 VBX 控制库中的控制参考指南中可以找到)。
- title 是控制的主题或标题。
- x 和 y 是父对象中要放置控制的位置点坐标。

■ w 和 h 是控制的宽度和高度。

■ module 是作为 TModule 的构造函数的参数传给类 TControl 的基本构造函数。它的默认值是0)。

第二个构造函数用在应用程序资源文件中定义的 VBX 控制设置一个 TVbxCantorol 对象。

```
TVbxControl(TWindow * Parent,
            int resId,
            TModule * module);
```

其中:

■ parent 是指向控制父类的一个指针。

■ resId 是在资源文件中 VBX 控制的资源 ID。

■ module 是作为 TModule 的构造函数的参数,传给类 TControl 的基本构造函数。

4.2.3 隐含地和显式地构造

可以显式地或隐含地构造 VBX 控制。当调用其中一个构造函数时则显式地构造一个对象;当不调用构造函数,而允许控制作为其父类的实例生成时,则隐式地构造一个对象。

隐式地构造对象包含调用 VBX 控制对象的一个构造函数,这通常在父类的构造函数中完成,所以当父窗口创建时,VBX 控制已经构造并作好准备。也可以在需要时才构造控制,例如,如果只有一个控制的空间时,可能就希望这样做。这种情况下,可能让用户选择一个菜单或按下一个按钮。然后,根据用户的动作,可能要生成一个对象的实例并在已存在的界面元素中显示它。下面的代码说明在一个对话框对象的构造函数中,用两个 TVbxControl 构造函数显式地构造控制。

```
Class TTestDialog::Public TDialog;public TVbxEventHandler
{ public:
    TTestDialog(TWindow * parent,char * name)
      : TDialog(parent,name),TWindow(parent,name);
    {
        new TVbxControl(this,IDCONTROL1);
        new TVbx(Control(this,IDCONTROL2,
                        "SWITCH. VBX","BiSwitch",
                        "& Program VBX Control",
                        16,70,200,50);
    }
    DECLARE_RE#SPONSE_TABLE(TTestDialog);
};
```

隐式地构造过程发生在应用程序的源代码外面,设计界面元素时,例如在 Resoure-Workhop 中。元素中增加 VBX 控制,然后在生成父类对象的具体事件时,其子类象编辑框、列表框、按钮、和 VBX 控制,便随父类一起自动生成。下面的例子显示了上述思想的代码。

值得一提的是,在下面的代码中看不到 VBX 控制,取而则代之的是,VBX 控制包含在对话资源 DIALOG_1中,当 DIALOG 被调用和生成时,VBX 控制便自动生成。

```
Class TTestDialog: public TDialog,public TVbxEventHandler
{
public :
    TTestDialog(TWindow * parent,Char * name)
      :TDialog(parent,name),TWindow(parent,name){}
    DECLARE_RESPONSE_TABLE(TTestDialog);
};
void TTestWindow::CmAbout()
{
    TTestDialog(this,"DIALOG_1"),Execute();
}.
```

4.2.4 类 TVbxEventHandler

类 TVbxEventHandler 非常小,在大多数程序中很少被人感兴趣,尽管如此,它完成的工作很重要,如果没有包含在 TVbxEventHandler 类中的功能,就不能同自己的 VBX 控制通信。事件处理程序模型在后续章节有详细描述,下面的部分只解释 TVbxEventHandler 是由一个单独的函数和一个消息响应表组成,这个函数是 EvVbxDispatch,它是处理事件的子程序,它调用名叫 WM_VBXFIREEVENT 的消息,将没有分解的消息转换到结构 VBX-EVENT 中,并分配一个新的消息,它是由控制的父类完成,因为父类对象必须从 TVbx-EventHandler 派生出,这就意味着父类用一个没的消息回调自身,新的消息更容易处理和理解,这就是后面部分描述的宏 WM_VBXEVENTNAME 所要处理的消息。

4.3 处理 VBX 控制消息

处理 VBX 控制消息是通过控制的父类对象实现的,为了使父类对象能处理这些消息,它必须是类 TVbxEventHandler 的派生类,为了做到这一点,可以将用到包含 VBX 控制的界面对象类(例如,TDialog,TFrameWindow 或从 ObjectWindows 界面对象类派生来的类)与类 TVbxEventHandler 结合。

4.3.1 事件响应表

一旦派生一个新类,就必须为其建立响应数。这个类的响应表与通常的响应表没有什么两样,这里仍然要处理所有命令消息和事件,像通常一样。唯一例外是宏 EV_VBXEVENT-NAME 要处理 VBX 控制中新的类消息。

宏 EV_VBXEVENTNAME 有三个参数:

```
EV_VBXEVENTNAME(ID,TEvent,EvHandler)
```

其中:

- ID 是控制 ID，可以在资源文件中两个构造函数的第二个参数或资源 ID 中找到它。
- Event 是一个标明事件名的串，它依赖于控制类型，可以是一个标准的 VBX 事件名或一个客户事件名。如果控制来源于第三部分的 VBX 控制库，则可以在控制参考指南中找到这个事件名。
- EvHandler 是事件和控制的处理函数，函数 EvHandler 定义如下：

```
void EvHandler(VBXEVENT FAR * event);
```

通过父类，从一个 VBX 控制中接受一条消息后，这条消息就被分配给控制和事件相应的函数。调用函数时，通过一个指针将信息传给结构 VBXEVENT，这个结构将在后面详细讨论。

4.3.2 解释一个控制事件

一旦一个 VBX 控制事件发生而且这个事件的处理函数调用后，这个函数就需要处理作为一个参数接受的 VBXEVENT 结构。

这个结构如下：

```
struct VBXEVENT {
    HCTL      hCtl;
    HWND      hWnd;
    int       n ID;
    int       iEvent;
    LPCSTR    lpszEvent;
    int       cParams;
    LPVOID    LpParams;
};
```

其中：

- hCtl 是发现 VBX 控制的句柄(不是窗口句柄)。
- hWnd 是控制窗口的句柄。
- nID 是 VBX 控制的 ID。
- iEvent 是事件索引。
- LpszEvent 是事件名。
- cParams 是本事件的参数数目。
- LpParams 是一个指针，它指向一个数组，这个数组由指向由本事件的参数值的指针组成。
- 为了理解这个结构，就必须理解 VBX 控制事件是如何工作的。刚开始的三个数据成员作用很明显：它们标识发出的控制消息。其后的两个数据成员也很简单，每个 VBX 控制发出的事件都有一个事件索引，即结构中成员 iEvent 和一个事件名，即结构中的成员 lpszEvent。

剩下的两个随同事件一起传递参数的数据成员就比较复杂了。参数 cParams 包含了本事件所需的参数数目，参数 lpParams 是一个数组，这个数组是由指向事件参数的指针组成

(像其他的数组一样,这个数组是从0开始到 cParams-1的,这两个参数比前面的几个参数都要复杂,因为关于每个参数的类型或含义都没有固定的指明,如果控制是第三部分的 VBX 控制库中得到的,可以在关于控制的 参考指南中找到这条消息,否则,必须从控制类的设计者那里获得信息(或者自己设计控制)。

4.3.3 发现事件信息

一般地,是通过参考 VBX 控制的文档来解释从一个事件中返回的消息的,如果这种方法行不通,TVbxControl 提供了一些获取关于事件信息的方法。可以通过调用 TVbxControl 的成员函数 GetNumEvents 来得到一个控制能发出的事件总数目,函数返回一个表示事件总数的整型数,这些事件是从0到函数 GetNumEvents 返回值减1之间排列的,函数 GetEventName 有一个参数,是表示检索数目的 int 类型值,返回一个包含事件索引相应的 int 类型值。

4.4 访问 VBX 控制

有两种方法可以直接问 VBX 控制,第一种方法是获得和设置控制的属性,每个控制都有固定数目的属性可以设置,从而改变其形式和性能。另一种访问 VBX 的途径是调用控制的方法,控制的方法近似于类中的成员函数,实际上是通过类 TVbxControl 中的成员函数来访问的,可以使用这种方法调用对象并产生一个动作。

4.4.1 VBX 控制属性

每个控制有都一些属性,控制的属性影响控制的形式及其行为,例如,控制中各个部分的颜色,控制的尺寸和位置,控制的主题,等等。改变这些属性是操作 VBX 控制类的主要方法。每个控制的属性都应在第三方软件的 VBX 控制库的参考指南里列出文档。如果一个控制不是第三方软件的控制或者是商业控制包的一部分,则必须向控制的设计者询问关于控制属性的限制和特殊含义。许多属性通常按索引方式起作用,例如,关于控制的背景模式:0表示不着色,1表示交叉阴影线,2表示黑色,等等。如果没有适当的关于控制的文档或信息,很难使用控制的属性。

4.4.2 发现控制属性信息

发现关于控制属性信息的方法通常是参考 VBX 控制的文档,如果这样还不行的话,类 TVbxControl 提供了一些获取关于控制属性的方法。可以通过调用类 TVbxControl 的成员函数 GetNumPreps,获取控制属性的数目,它返回一个整型(int)值,记载了属性的总数目,这些属性是从0到函数 GetNumProps 返回值减1之间索引的,可以通过调用类 TVbxControl 成员函数 GetPropName 获得其中的属性名,函数 GetPropName 有一个参数,一个 int 类型的索引号,返回一个包含属性名的串。

反之,也可以通过调用 TVbxControl 的成员函数 GetPropIndex 获得属性的索引,函数 GetPropIndex 有一个参数,一个包含属性名的串,返回一个与属性索引相应的 int 类型值。

4.4.3 获得控制属性

可以通过属性的名字或其索引号获得一个控制属性的值，虽然使用索引在某种程度上来说更有效(因为不必查询一个串)，但使用属性名通常显得更直观，根据具体情况可以选择任一种方法。

类 TVbxControl 提供了函数 GetProp 获取控制的属性，加载函数 GetProp，允许使用索引号和属性名获取属性，下面不同版本的函数的进一步调用允许获取多个不同类型的属性。

```
// get properties1by index
BOOL GetProp(int propIndex,int& value,int arrayIndex=-1);
BOOL GetProplint propIndex,long& vlalue,int arrayIndex=-1);
BOOL GetProp(int propIndex,HPIC& value,int arrayIndex=-1);
BOOL GetProp(int propIndex,float& valut,int array Index=-1);
BOOlGetProp(int PropIndex,string& value,int array Index=-1);
// get Properties by name
BOOL GetProp(const char far * name,int& value,int arrayIndex=-1);
BOOL GetProp(const char far * name,long& value,int arrayIndex=-1);
BOOL GetProp(const char far * name,HPIC& value,int arrayIndex=-1);
BOOL GetProp(const char far * name,float& value,int arrayIndex=-1);
BOOL GetProp(const char far * name,string& value,int arrayIndex=-1);
```

在第一个参数是 int 类型的版本里，是通过传递属性索引，确定属性的；在第一个参数是 Char＊类型的版本里，是通过传递属性名，确定属性的。函数 GetProp 的第二个参数是属性数据类型的参考，而不是返回属性的值，创建一个与属性相同类型的对象，并将这个参考值传给该对象的 GetProp 函数，函数 GetProp 返回时，对象包含属性的当前值。

第三个参数是属性数组的索引，如果控制需要就必须提供，可以通过阅读 VBX 控制的文档知道是否需要提供这个参数和要求的值，如果是－1，函数就忽略这个参数。

4.4.4 设置控制的属性

就像获取控制属性一样，也可以通过控制的属性名或索引号设置控制属性值，虽然使用索引在某种程度上来说更有效(因为不必查询一个串)，但使用属性名通常显得更直观，根据具体情况可以选择任一种方法。

类 TVbxControl 提供了函数 SetProp 设置控制的属性，加载函数 SetProp，允许使用索引号和属性名设置属性，下面不同版本的函数的进一步调用允许设置多个不同类型的属性。

```
// Set properties by indexBOOL SetProp(int propIndex,int value,int arrayIndex=-1);
BOOL SetProp(int propIndex,long value,int arrayIndex=-1);
BOOL SetProp(int propIndex,float value,int arrayIndex=-1);
BOOL SetProp(int propIndex,const char far * value,int arrayIndex=-1);
BOOL SetProp(int propIndex,const String& value,int arrayIndex=-1);
BOOL SetProp(int propIndex,const char far * value,int arrayIndex=-1);
// Set properties by name
BOOL SetProp(const char far * name,int value,int arrayIndex=-1);
```

```
BOOL SetProp(const char far * name,long value,int arrayIndex=-1);
BOOL SetProp(const char far * name,HPIC value,int arrayIndex=-1);
BOOL SetProp(const char far * name,float value,int arrayIndex=-1);
BOOL SetProp(const char far * name,const string& value,int arrayIndex=-1);
BOOL SetProp(const char far * name,const char far * value,int arrayIndex=-1);
```

在第一个参数是 int 类型的版本里，是通过传递属性索引，确定属性的；在第一个参数是 char * 类型的版本里，是通过传递属性名，确定属性的。第二个参数就是属性将要设置的值。第三个参数是属性数组的索引，如果控制需要就必须提供，可以通过阅读 VBX 控制的文档知道是否需要提供这个参数和要求的值，如果是－1，函数就忽略这个参数。

虽然有五种不同的数据类型可以传给函数 GetProp，但 GetProp 可以传递六种数据类型。这是因为最后两个版本的函数用了 char * 类型和 ANSI 串类表示了一个串，这在向一个控制传递字符型串就时有了更多的弹性。

4.4.5 VBX 控制方法

VBX 控制方法是一些包含在每个 VBX 控制中的函数，可以在控制中调用并产生一些动作，TVbxControl 控制与 Visual Basic 1.0中包含的控制具有一定兼容性：

```
Move(int x,int y ,int w,int h);
Refresh();
AddItem(int index,const char far * item);
RemoveItem(int index);
```

函数 Move 将控制移到坐标 x,y 处，并将控制尺寸修改为 w 点宽和 h 点高。

函数 Refresh 刷新控制 显示区。

函数 AddItem 在控制的项目表中增加一条项目 item，并给出新的项目索引号 index。

函数 RemoveItem 除去具有索引号 index 的一项。

第五章　应用举例

5.1　第一步：设计一个简单的应用程序

本节从一个典型的范例开始，打开文件 STEP01. CPP，它显示了许多关于 ObjectWindows 应用的有用知识。由于程序较短，这里将整个文件显示如下：

```
//---------------------------------------------------------------
//  ObjectWindows-(c) Copyright 1991,1993 by Borland International
//  Tutorial application--step01. CPP
//---------------------------------------------------------------
#include <owl\applicat. h>
#include<owl\framewin. h>
    Class TMyApp:public TApplication
    { pbulic:
        TMyApp():TApplication(){}
        void InitMainWindow()
        {
        SetMainWindow(new TFrameWindow(0,"Sample ObjectWindows
                        Program"));
        }
};
int OwlMain(int /* argc */,char * /* argv */[])
    { return TMyApp(). Run();}
```

这个简单的应用程序包含了许多重要的特点：

■ 源文件中包含两个头文件 OWL\applicat. h 和 Owl\famewin. h。包含这两个头文件是因为应用程序中用到了 TApplication 和 TFrameWindow 这两个 ObjectWindows 类。今后，凡是程序中用到了 ObjectWindows 类，就必须包含相应的头文件，以便程序能正确的编译。

■ 类 TMyApp 是 ObjectWindows 的类 TApplication 的派生类。每个 ObjectWindows 应用程序都有一个 TApplication 类对象——或者更普遍一点说，有一个 TApplication 派生类对象，——通常被认为是应用程序对象。如果试图直接使用 TApplication 类对象，会发现很难引导程序流运行。启动 TApplication 类可以访问应用程序对象的工作，还可以加载必须的函数，使应用程序按设计者的思路工作。

■ 除了应用程序对象外，每个 ObjectWindows 应用程序有一个 OwlMain 函数，应用程序对象实际上是在 OwlMain 函数中用一个简单的声明定义的 OwlMain 函数在 ObjectWindows 程序中的作用相当于 WinMain 函数在一般的 Windows 应用程序中的

作用。可以用 OWLMain 检查命令行参数,设置全局数据,或完成在应用程序执行前你关心的其他任何工作。

- 为了执行应用程序,先调用应用程序对象的 Run 函数,Run 程序首先调用 InitApplication 函数,不过只有在应用对象的事件是第一个事件时才是这样(默认状态下,TApplication::InitApplication 函数不起作用),在 InitApplication 函数返回之后,Run 函数调用 InitInstance 函数,它初始化应用程序对象的每一个事件。默认情况下,TApplication::InitInstance 调用函数 InitMainWindow,它初始化应用程序的主窗口,然后创建和显示主窗口。
- TMyApp 加载 InitMainWindow 函数。如果愿意,可以用这个函数设计主窗口。函数 SetMainWindow 将应用对象的主窗口设置给 TFrameWindow 类或 TFrameWindow 导出类的对象。在这种情况下,程序简单地创建一个新的没有父类的 TFrameWindow 窗口(TFrameWindow 的第一个参数是一个指向父类的指针)和标题 Sample ObjectWindows Program。

这个简单的应用程序介绍了 ObjectWindows 程序设计中两个非常重要的概念。像看到的那样明显,从 TApplication 类导出一个类和启动 InitMainWindows 函数,能获得很多功能控制程序的运行。以后的章节中会发现,从这个简单的应用程序中获得的技能将会使你在庞大和复杂的程序设计中匠心独具。

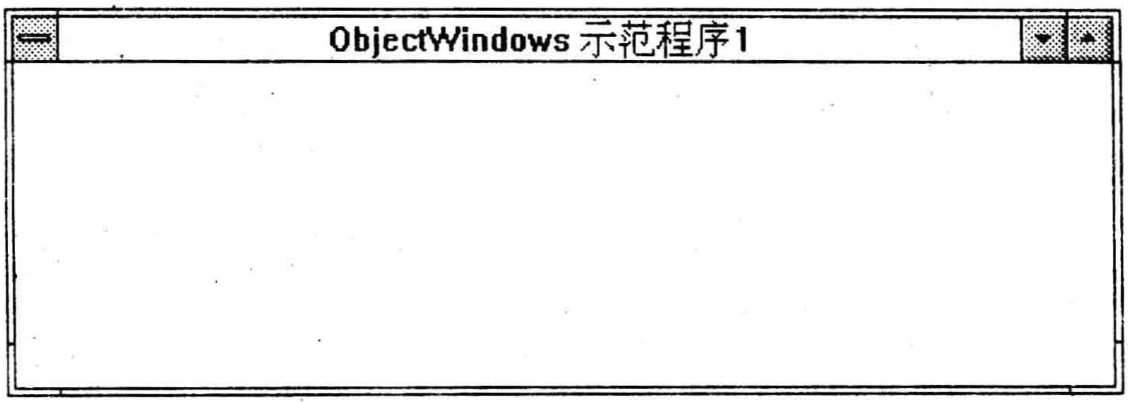


图5.1 一个简单的 ObjectWindows 程序

5.2 第二步:处理窗口事件

本节所涉及的源程序在目录 EXAMPLE\OWL\TUTORIAL 下的文件 STEP02.CPP 中可以找到。

这一节介绍响应表和 ObjectWindows 的另一个重要属性。在 ObjectWindows 2.0的应用程序中,响应表控制事件和消息处理,将事件分配给相应的事件处理函数。第二节里还增加了这些事件处理函数。

5.2.1 增加窗口类

在使用了名叫 TMyWindow 类的应用程序中增加了响应表。TMyWindow 类是 TWin-

dow 类的导出类。定义如下：

```
Class TMyWindow:public TWindow
{
    public:
        TMyWindow(TWindow * parent=0);
    protected:
    // override member function of TWindow
        BOOL CanClose();
    // message response functions
    void EvlButton Down(UINT,TPoint&);
    void EvRBUtton Down(UINT,TPoint&);
    DECLARE_RESPONSE_TABLE(TMyWindow);
};
```

这个类的构造函数非常简单，其中只有一个参数，是 TWindow 类型，它指向对象的父窗口。构造函数的定义如下：

```
TMyWindow::TMyWindow(TWindow * parent)
{
    Init (parent,0,0);
}
```

Init 函数初始化 TMyWindow 类的的基类。在这种情况下，其调用并不复杂。根据程序设计的需要，唯一要做的可能是对窗口的父类的调用，而且你会发现，即使这一点，程序也替你考虑了。

5.2.2 增加响应表

TMyWindow 类中唯一的公共成员是其构造函数。但是，如果其他的成员是保护类型的，怎么去访问它们呢？答案在于响应表的定义。注意 TMyWindow 类定义的最后一行，定义了一个响应表，也就是说，它告诉类有一个响应表，非常像函数在类中声明自己的存在一样，但是它不定义函数的具体实现。

响应表的定义设置类去处理窗口事件，并将每个事件送给相应的事件处理函数，按一般的规律，事件处理函数应该是保护类型的，这能防止类和函数在其定义的类处理调用。这里是 TMyWindow 类响应表的定义。

```
DEFINE_RESPONSE_TABLE(TMyWindow,TWindow)
    EV_WM_LBUTTONDOWN,
    EV_WM_RBUTTONDOWN,
END_RESPONSE_TABLE;
```

可以将响应表放在源文件的任何地方。

现在，可以把响应表做得非常简单，下面是对表中每一部分的描述，一个响应表含有四个重要的组成部分：

- 在类定义中声明响应表。
- 响应表第一行的定义通常是宏 DEFINE_RESPONSE_TABLEX。X 的值依赖于类的继承关系，它取决于类的直接基类的数目，这里，TMyWindow 类只有一个直接基类 TWindow。
- 响应表的最后一行的定义通常是宏 END_RESPONSE_TABLE，结构事件响应表的定义。
- 在宏 DEFINE_RESPONSE_TABLEX 和宏 END_RESPONSE_TABLE 之间的是将特定的事件和相应处理函数相联系的一些宏。

本例响应表中间的两个宏 EV_WM_LBUTTONDOWN 和 EV_WM_RBUTTONDOWN 是为标准的窗口消息 WM_LBUTTONDOWN 和 WM_RBUTTONDOWN 定义的。所有的标准窗口消息都有 ObjectWindows 定义的响应表的宏。为了寻找特定的消息宏的名字。消息名前面加上 EV_ 前缀，例如，处理 WM_PAINT 消息的宏为 EV_WM_PAINT，而处理 WM_LBUTTONDOWN 消息的宏是 EV_WM_LBUTTONDOWN。这些预定义的宏将消息传给用这些预定义名定义的函数。决定函数名时，首先用 EV 替换 WM_，然后将每个单词除首字母外的大写字母变成小写字母，除去单词之间的下划线形成一个单词。例如，消息 WM_PAINT 传给名叫 EVPaint 的函数，消息 WM_LBUTTONDOWN 传给名叫 EvLButtonDown 的函数。

5.2.3 事件处理函数

读者可以看到，在类 TMyWindow 中有两个保护类型的函数 EvLButtonDown 和 EvRButtonDown。由于响应表中宏的定义，当类 TMyWindow 接受一个 WM_LBUTTONDOWN 事件或 WM_RBUTTONDOWN 事件时，就会将其传给相应的事件处理函数。处理事件 WM_LBUTTONDOWN 和 WM_RBUTTONDOWN 的函数很简单，每一个函数弹出一个消息框显示用户已经摁下某个键。这个函数的源代码应该是这样的：

```
void TMyWindow::EvLButtonDown(UINT,TPoint&)
{
    MessageBox("You have pressed the left mouse button",
               "Message Dispatched",MB_zok0;
}
    void TMyWindow::EvRButtonDOwn(UINT,TPoint&)
{
    MessageBox("You have pressed the right mouse button",
               "Message Dispatched",MB_OK);
}
```

这里显示了 ObjectWindows 2.0处理标准窗口事件的非常优秀的特点，处理每个事件的函数能接受任意的参数类型(所有的宏和其相关函数都在 ObjectWindows Reference Guide 一书中的第二章中列出)。事实上，这些参数类型的信息经过计码存放在变量 WPARAM 和 LPARA 中，通常和事件一起传送，事件信息自动“分解”。

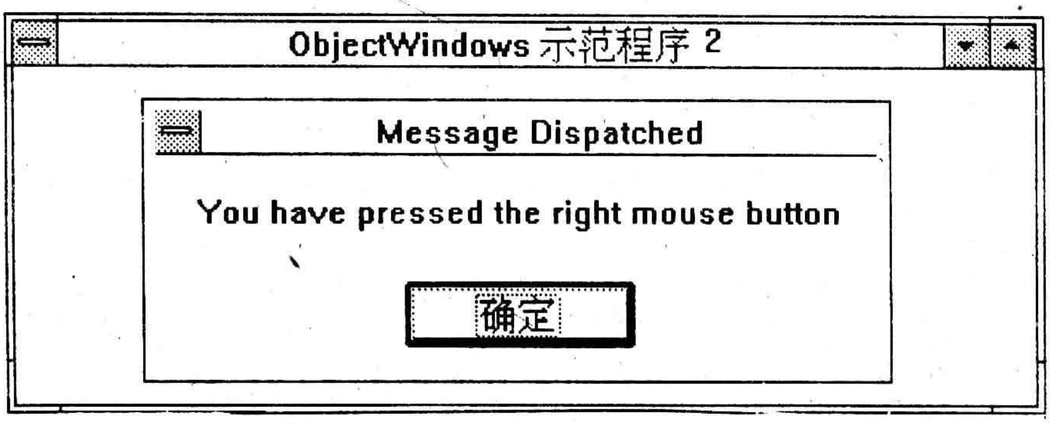


图5.2 范例程序响应一个用户事件

这种方法的优点有两个方面：

- 程序设计者可以不再手工从 WPARAM 和 LPARAM 值中提取信息。
- 预定义函数允许编译时的类型检查，还可以防止出现追踪困难的错误，这种错误可能在混淆变量 WPARAM 和 LPARAM 译码值时发生。

例如，消息 WM_LBUTTONDOWN 和 WM_RBUTTONDOWN 都包含变量 WPARAM 和 LPARAM 的相同类型的信息：

- WPARAM 包含键的标志，它们记录用户是否揿下了虚拟键中的某一个。
- LPARAM 的低位字记录光标的 x 坐标。
- LPARAM 的高位字记录光标的 y 坐标。

函数 EvLBUttonDown 和 EvRButtonDown 也有相似的标志，每个函数的 UINT 参数都与键标志参数相关。通常在 LPARAM 中译码的值取代了存储在 TPoint 对象中的值。

5.2.4 封装的 API 函数调用

读者可能注意到对函数 MessageBox 函数的调用显得有点奇特。Windows 的 API 函数 MessageBox 将 HWND 作为它的第一个参数，但是在这里调用的 MessageBox 函数实际上是类 TWindow 的一个成员函数，有许多函数都是这样的：它们有和 Windows 的 API 函数相同的名字，但是它们的标号是不同的，最普通的区别是除去 HWND 和 HINSTANCE 这样的句柄参数，用 ObjectWindows 数据类型替代 Windows 数据类型等等。在这种情况下，窗口类提供 HWND 参数。

5.2.5 重写 CanClose 函数

TMyWindow 类的另外特点在于 CanClose 函数。在一个应用程序试图关闭一个窗口前，它要调用 CanClose 函数，如果返回为假，则窗口忽略关闭，反之，如果返回为真，则关闭窗口。

从应用程序的角度看，这样做能保证正在使用的或尚有未存储数据的窗口不被随意关闭；从窗口的观点来看，应用程序在关闭窗口之前，向用户提出警告，还能给用户机会让其在关闭窗口之前对窗口内的东西进行清理。

这里是 TMyWindow 类中的 CanClose 函数：

```
BOOL TMyWindow::CanClose()
{ Return MessageBox("Do you Want to save?","Drawing
  has changed",MB_YESNO|MB_ICONQUESTION==IDNO);
}
```

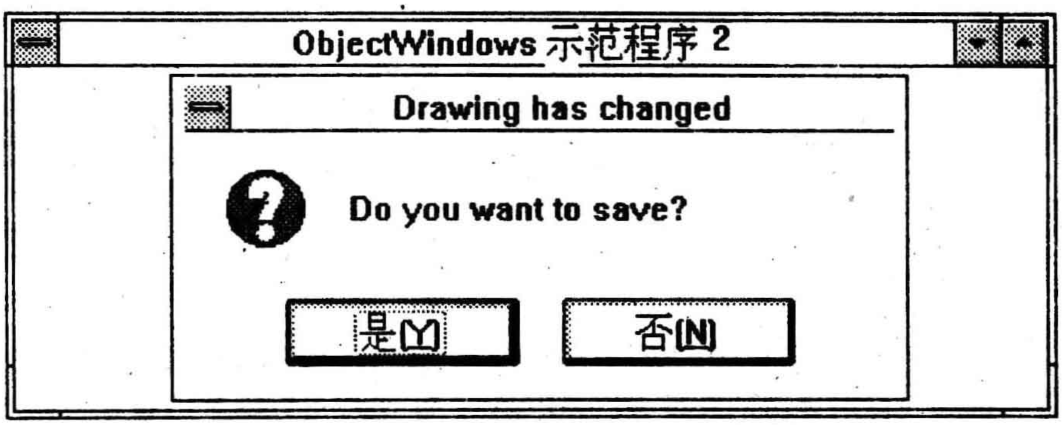


图5.3 改进关闭进行为以后的范例程序

此处，函数仅仅弹出一个信息框表示画面已被改变，询问用户是否保存画面，因为在这里没有画面可以存储，这个消息实际上是不起作用的，但是到了第七节，当程序中增加了向文件中存储数据的功能后，就会有用了。

5.2.6 将 TMyWindow 作为主窗口

最后要做的事就是实际地创建一个新的 TMyWindpw 类的实例。读者可能认为要实现这一目的，只要在函数 InitMainWindow 定义中对函数 SetMainWindow 的调用 TMyWindow 类替代 TFrameWindow 类就可以了：

```
void InitMainWindow(),
{ SetMainWindow(new TMyWindow);
}
```

事实上，这是行不通的，原因很多，但是主要的原因是因为 TMyWIdow 类不是 TMy-Window 类，使其基于 TFrameWindow 类而不是 TWindow 类，虽然这很容易做到，但是它将实用性引入 TMyWindow 类，并不是必须的。以后的章节中会发现，TMyWindow 为在有一个独特的用途。增加类 TMyWindow 的框架性能，将会削弱其可变性。

第二种方法是类 TFrameWindow 中将类 TMyWindow 对象当作一个客户使用，这很容易做，利用已经使用的 TFrame Window 类的构造函数的第三个参数定义一个 TWindow 类或 TWindow 派生类的对象，把它作为框架的一个客户。其代码如下：

```
SetMainWindow(new TFramewineow(0,"Sample ObjectWindows Program",
                              new TMyWindow));
```

通过这种图径，TFrameWindow 类管理框架窗口，而让 TMyWindow 类自由地管理自己的任务，这使得对类对象的设计更完善和模式化。这样做还能很容易地改变已经使用的框架

窗口的类型，这在以后将会看到。

注意：*TrameWindow* 构造函数中，*NewTMyWindow* 还没有为 *TMyWindow* 对象指定父类。因为到目前还没有父类。*TFrameWindow* 对象是父类，但到目前还没有构造该类；一旦创建了 *TFrameWindow* 对象，它会自动将自己设为客户窗口的父类。

5.3 第三步：在窗口中绘制文本

本节讲述的源程序在 EXAMPLE\OWL\TUTORIAL 目录下的 STEP03. CPP 中可以找到。

这一节从第二节应用程序增加的新窗口开始，当鼠标按钮被按下时，不再弹出一个消息框，事件处理函数具备一些真正的功能——按下左键，则显示触发鼠标键时鼠标在窗口中所在位置点的坐标，按下鼠标右键，窗口被刷新。

这个新增功能的代码在函数 EvlButtonDown 中，传给函数 EvLButtDown 的参数保存着鼠标被按下时的坐标，还必须在函数中增加一个字符串保存该处的文本。可以用函数 wsPrintf 去格式化这个字符串。现在可以设置窗口输出字符串了。

5.3.1 构造设备上下文

为了实现窗口中各种类型的图形操作，必须有一个窗口或工作区域的设备上下文。ObjectWindows 提供了许多类使得很容易设置、使用和清除设备上下文。因为类 TMyWindow 在框架窗口中是作为一个客户工作的，所以要使用 TClientDc 类，TClientDc 是类 TDc 的派生类，包含在头文件 OWL\dc. h 头文件中定义。

TClientDC 类有一个构造函数，它将 HWND 作为其唯一的参数，由于需要一个设备上下文作为 TMyWindow 类的对象，所以必需提供那个窗口的句柄。幸运的是，TWindow 基类提供一个 HWND 转换操作符。只要在要求一个 HWND 句柄的地方要使用窗口对象时，这个操作符就被隐含的调用。因此类 TClientDC 对象的构造函数是这样的：

```
TCLientDC dc( * this);
```

这里 * this 不是引用，而是窗口的句柄，HWND 转换符不对窗口对象指针起作用。

5.3.2 在设备上下文中输出文本

一旦建立了设备上下文，实际上就可以输出字符串了。类 TDC 提供了好几个 TextOut 函数的版本，像第二节的 MessageBox 函数，函数 TextOut 包含在设备上下文类中，和 Windows 的 API 函数 TextOut 非常相似。除了省略了第一个参数 HDC，函数 TextOut 的第一版与 Windows API 版的 TextOut 相差无几。

```
virtual BOOL TextOut(int x, int y, const char far * str,
                     int count=-1);
```

HDC 参数被 TDC 对象填充，第二版的 TextOut 函数省去了 HDC 参数，并将 x 和 y 的

坐标值组合成一个单个的 TPoint 结构：

```
BOOL TextOut(const TPoint& p ,const char far * str,
             int count=-1);
```

由于坐标是通过一个 TPoint 对象传到 EvLButtoDown 函数，所以可以用第二版的函数 TextOut 在窗口中打印坐标值，完整的 EvLButtonDown 函数如下：

```
void TMyWindow::EvlButtonDown(UINT,TPoiint& point)
{ char s[16];
  TClientDC dc( * this);
  wsprintf(s,"(%d,%d)",point.x,point.y);
  dc.TextOut(point,s,strle(s));
}
```

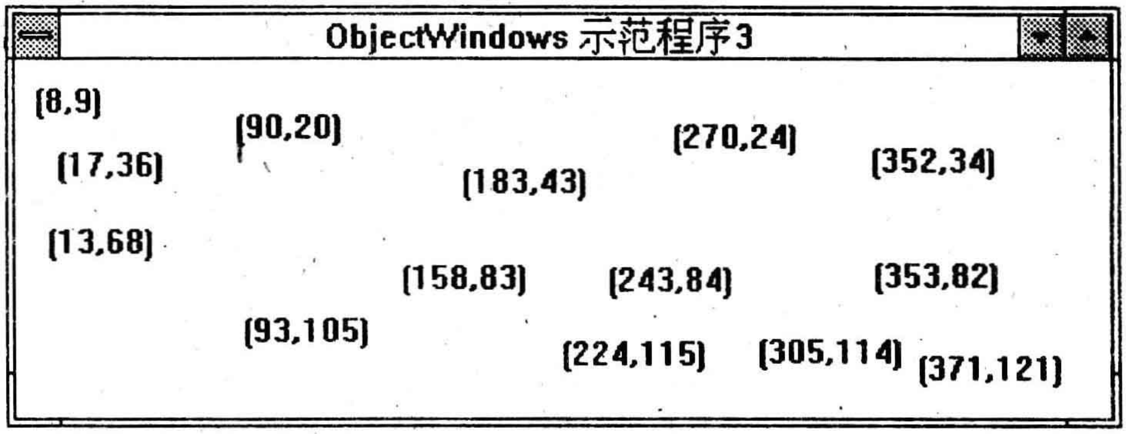


图5.4 在用户触发鼠标处显示文本

由于使用了 strlen，所以在程序中要加上头文件 string.h。。

5.3.3 清除窗口

类 TMyWindow 的基类 TWindow 提供了三个不同的失效函数，其中的两个函数 InvlaidateRect 和 InvalidateRgn 形式和功能上都像其 Windows 的 API 版本，但是都省略了参数 HWND。第三个函数 Invalidate 使窗口中的整个客户区失效。函数 InvlaLidate 只有一个 BOOL 类型的参数，它表示窗口刷新时无效区域是否擦除，默认状态下参数值为真。

因此，要擦除整个 TMyWindow 窗口的客户区域，只须调用函数 Invalidate，其参数要么取真，要么什么也没有。当用户按鼠标右键时，要清除屏幕，就必须在函数 EvRButtonDown 中进行调用。这个函数是这样定义的：

```
void TMyWindow:EvRButtonDOwn(UINT,TPoint&)
{ Invalidate();
}
```

5.4 第四步：在窗口中画线

本节的实例参见 EXAMPLES\OWL\TUTOPIAL 目录下的 SETP04.CPP 文件。

在这一节，将在窗口中增加画线的功能，画线是通过按下鼠标左键和拖动实现。为实现这一功能，在类 TMyWindow 的响应表中必须增加两个事件，WM_MOUSEMOVE 和 WM_LBUTTONUP，连同处理这些事件函数。还要在类中增加 TClientDC * 。

5.4.1 增加新的事件

为了让用户能在窗口中画线，应用程序必须处理一系列事件：

- 为了画线，程序必须监视用户什么时候按了鼠标左键。这已在处理 WM_LBUTTONDOOWN 事件时作了考虑。
- 一旦用户按了鼠标左键，就必须监视鼠标的移动，从这一点开始划线。要想知道什么时候用户移动了鼠标，捕捉事件 WM_LBUTTONUP 即可。
- 程序必须知道什么时候用户完成了画线操作，鼠标左键释放，就意味着操作结束。可以通过捕捉事件 WM_LBUTTONUP 来监视它。

在窗口类的响应表中，必须增加两个宏 EV_WM_MOUSEMOVE 和 EV_WM_LBUTTONUP 新的响应表应该是这样定义的：

```
DEFINE_RESPONSE_TABLE1(TMyWindow,TWindow)
    EV_WM_LBUTTONDOWN,
    EV_WM_RBUTTONDOWN,
    EV_WM_MOUSEMOVE,
    EV_WM_LBUTTONUP,
END_RESPONSE_TABLE;
```

还必须将函数 EvLButtonUp 和 EvMouseMove 加入类 TMyWindow。

5.4.2 增加一个 TClientDC 指针

第三步画线的方案不是令人特别满意。

- 第三节，在函数 EvLBUttonDown 中创建了 TClientDC 对象，当函数返回时会自动清除，但是现在需要一个有效的设备上下文贯穿在三个不同的函数 EvLButtonDown，EvMouseMove 和 EvLButtonUp 中。
- 你可以捕捉 WM_MouseMove 事件，并且从当前点画线一直画到函数 EvMouseMove 处理得到的最终点。但是当鼠标运动时事件 WM_MOUSEMOVE 被释放，用户只能画一条鼠标被按下并拖动经过的轨迹线。

思考一下这两个问题，非常简单，通过在类 TMyWindow 中加入一个新的保护型数据成员，便会令人满意，这个数据成员中一个叫 DragDC 的 TDC * 类型变量。

它是这样起作用的：

- 当鼠标左键被按下，调用函数 EvLButtonDown 函数。这个函数产生新的 TClientDC 并赋给 DragDC，然后将当前在 DragDC 中的点值设置到鼠标被触发的位置。这段程序代码如下：

```
void TMyWindow::EvLButtonDown(UINT,TPont& Point)
```

```
    {
        Invalaidate();
        if(!DragDC){ SetCapture();
                     DragDC=new TClientDC(*this);
                     DragDC-> MoveTo(point);
        }
    }
```

■ 当鼠标左键被释放时，调用函数 EvLButtonUp。如果 DragDC 是有效的（也就是说，它表示着一个有效的设备上下文），EvLButtonUp 就删去它，并设置为0。这段程序代码如下：

```
void TMyWindo::EvLButtonUp(UINT,TPoint &)
    {
        if (DragDC) { ReleaseCapture();
                      delete DragDC;
                      DragDC=0;
        }
    }
```

■ 当鼠标移动时，调用函数 EvMouse Move，这个函数通过检查 DragDc 就是，判断鼠标左键是否被按下。如果 DragDc 是0，那么鼠标或者没有被按下或者按下又被释放；另一种方式是用户没有画线，函数返回。如果 DragDc 是有效的，意味着鼠标左键目前是被按下的，则调用 TWindow::LIneTo 函数从当前点画到新的一点。

```
void TMyWindow::EvMouseMove(UINT,TPoint * point)
{
    if(DragDC)
        DragDC->LineTo(Point);
}
```

一、初始化 DragDC

必须保证在构造 TMyWindow 类对象时将 DragDC 设置为0。

```
TMyWindow::TMyWindow(TWindow * parent);
{
    Init(parent,0,0);
    DragDC=0;
}
```

二、使用后清除 DragDC

由于变量 DragDC 是一个指向 TClient 类对象的指针，不是真正的 TClient 类对象，当 TMyWindow 对象清除时它不能自动被清除，所以必须在类 TMyWindow 中增加一个析构函数以便正确地清除它。析构函数要做的工作是在调用函数 delete 删除 DragDC。

类 TMyWindow 现在变成这样：

```
class TMyWindow:public TWindow
{
    public:
        TMyWindow(TWindow * parent=0);
        ~TMyWindow()(delet DragDC;);
    protected:
        TDC * DragDC;
        // Override member functionof TWindow
        BOOL CanCLose();
        // MESSAGE response functions
        void EvLButtonDown(UINT,TPoint &);
        void EvRButtonDown(UINT,TPoint&);
        void EvMouseMove(UINT,TPoint&);
        void EvLButtonUp(UINT,TPoint&);
        DECLARE_RESPONSE_TABLE(TMyWindow);
};
```

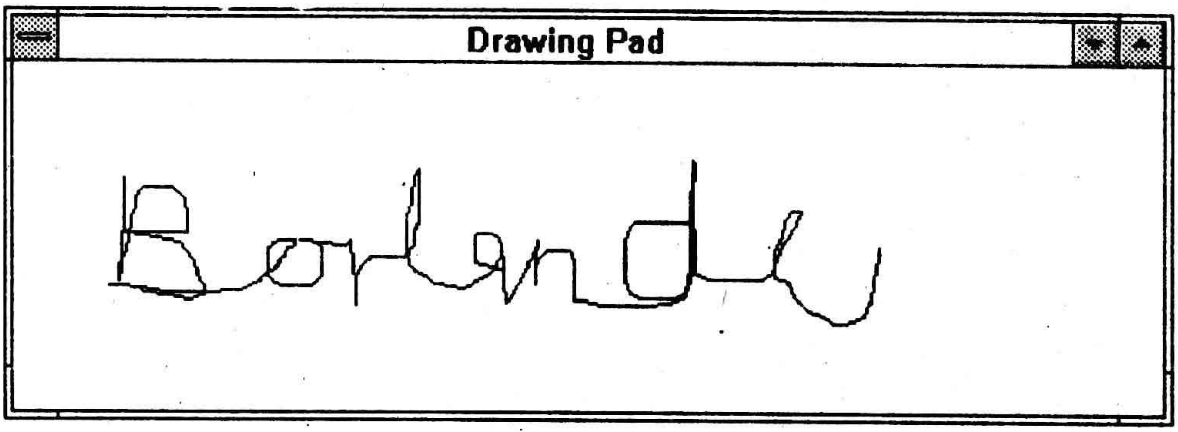


图5.5 范例程序在窗口中划线

注意：由于示范的应用程序现在变得有点用处了，所以主窗口的名字也从"*Sample Objectwindows Program*"变成"*Drawind Pad*"：*SetMainWindow(new TFrameWindow(0,"Drawing Pad",new TMyWindow))*；

5.5 第五步：改变画线宽度

在这一节中，你将学会如何进一步完善前面所述应用程序的绘图功能，即可以改变线的宽度。为了做到这一点，需要为 TMyWindow 类增加一个绘制对象 TPen * 及一个保存线的宽度值的 Int 类型数(整型数)。

5.5.1 增加画笔

为了给窗口类增加画笔，需要增加两个保护类型的数据成员(protected)：Pen(TPen * 类型)与 Pensize(整型)。增加画笔后带来的主要功能是由 EvLButtonDown 和 EvRButtonDown 两个函数来实现的。

一、初始化笔

在使用 Pen 对象与 penSize 进行绘制之前，用户必须进行创建与初始化操作，最好是在 TMyWindow 的构造函数中进行上述操作。

```
TMyWindow::TMyWindow(TWindow * parent)
{
    Init (parent,0,0);
    DragDC=0;
    PenSize=1;
    Pen=new TPen(TColor::Black,PenSize);
}
```

TPen 构造函数中的 TColor::Black 对象是在 OWL\Color.h 头文件中定义的一个 enum (枚举)型的数据。这里，笔被定义成黑色。在后面的第九节中，你会了解到关于 TPen 构造函数的更详细的知识。

二、把笔选入 DragDC

为了使用利用刚增加的笔画线，必须先把笔选入设备上下文。设备上下文类中有一个成员函数叫作 SelectObject，这个函数与 Windows API 中的函数 SelectObject 很类似，二者的区别的仅仅是前者不需要一个设备上下文句柄。

你可以通过 SelectObject 将一些不同类型的对象选入设备上下文，例如画笔、刷子、字体、调色板等。在开始画线之前调用 SelectObject 函数。例如在 EvLButtonDown 函数中，在创建了设备上下文后紧接着调用了 SelectObject 函数，如下所示：

```
void TMyWindow::EvLButtonDown(UINT,TPoint& point)
{
    Invalidate();
    if( ! DragDC){
        SetCapture();
        DragDC=new TClientDC( * this);
        DragDC->SelectObject( * Pen);
        DragDC->MoveTo(point);
    }
}
```

注意：在上面的 *SelectObject* 调用中，*Pen* 是以引用的方式调用的。这是因为 *SelectObject* 把 *TPen*&（对 *TPen* 类对象的引用）作为其参数，而且 *pen* 是一个 *TPen* * 对象（指向 *TPen* 类对象的指针）。通过对指针的引用使得 *Pen* 符合 *SlectObject* 函数参数类型的要求。

5.5.2 改变画笔尺寸

若能够改变画笔的尺寸并且能在程序中应用，就会很有意义。为此需要改变"按下鼠标右键"所代表的含义，即不再执行原来的清屏动作，而是表示用户要改变画笔的画线宽度。为此，这一处理过程需要在 EvRButtonDown 函数中完成。

当用户按下鼠标右键表示要改变画笔的画线宽度时，应找到一种能够使用户输入新的画笔尺寸的方法。为此，需要在屏幕上弹出一个输入对话框，在该对话框中，用户可以输入新的画笔尺寸。

一、构造输入对话框

TInputDialog 构造函数如下：

```
TInputDialog(TWindow * parent,
             const char far * title,
             const char far * prompt,
             char far * buffer,
             int buffersize
             TModule * module=0);
```

其中：

- parent 是指向对话框父窗口指针。在本例中，父窗口是 TMyWindow。也可以简单地以 this 指针来替代 parent 参数。
- title 和 prompt 是录对话框在屏幕上打开时，显示给用户的消息。在本例中，title（显示在对话框的标题栏）为"Line Thickness"，prompt（显示在对话框内部）为"Input a new thickness:"。
- buffer 为一个字符串。在执行 TInputDialog 之前，该字符串可以被初始化。如果 buffer 包含一个有效的字符串，它将被显示在对话框内作为用户的缺省响应。在本例中，用当前画笔的尺寸 PenSize 并初始化 buffer。
- bufferSize 是 buffer 的字节数。获取 bufferSize 的最简单的方法是用#define 预定义或者用 Sizeof(buffer)计算。
- Module 在本例中没有用到。

为了使用 TInputDialog，开发利用确保在源文件中及资源描述文件中包含 TInputDialog 的资源及资源标识符。它们包含在文件 include\OWL\inputdia. rc 中。因此，应当在你的资源文件及 C++ 源程序中包含资源文件 inputdia. rc。

二、执行输入对话框

一旦建立了一个 TInputDialog 对象,既可以通过调用 TDialog::Execute 函数来执行有模式的对话框,也可通过调用 TDialog::Create 函数来执行无模式的对话框。因为没有必要来执行无模式的对话框,所以可以使用 Execute 函数。

执行输入对话框的 Execute 函数结束后返回两个很重要的框,它们是 IDOK 和 IDCANCEL。究竟返回哪一个值取决于所按下的按钮,如果用户按下的是 OK 按钮,Execute 返回 IDOK;如果用户按下的是 Cancel 按钮,则 Execute 返回 IDCANCEL 故此,当执行一个输入对话框时,应该确保只有当返回值是 IDOK 时才能改变画笔尺寸。否则,仍保留原来的画笔尺寸。

如果 Execute 的返回值是 IDOK,新的画笔尺寸以字符串的形式在输入对话框的 buffer 中返回,在它能被应用之前,必须把它转换为一个整型数。同时还必须检查得到数值是否有效。最后,只有当所输入的数据检查有效,才能接受,并以此值来改变画笔尺寸。

TMyWindow 有一个成员函数叫作 SetPenSize,可以通过它来改变画笔尺寸,为什么这样做而不直接修改画笔,将在下一小节中详细讲解。

EvRBUttonDown 函数如下所示:

```
void TMyWindow::EvRButtonDown(UINT,TPoint&)
{
    char inputText[6];
    wsprintf(inputText,"%d",PenSize);
    if((TInputDialog(this,"Line THikness",
                    "Input a new thickness:",
                    inputText,
                    sizeof(inputText))).Execute()==IDOK){
        int new Pensize=atoi(inputText);
        if(newPenSize<0)
            newPenSize=1;
        SetPenSize(newPenSize);
    }
}
```

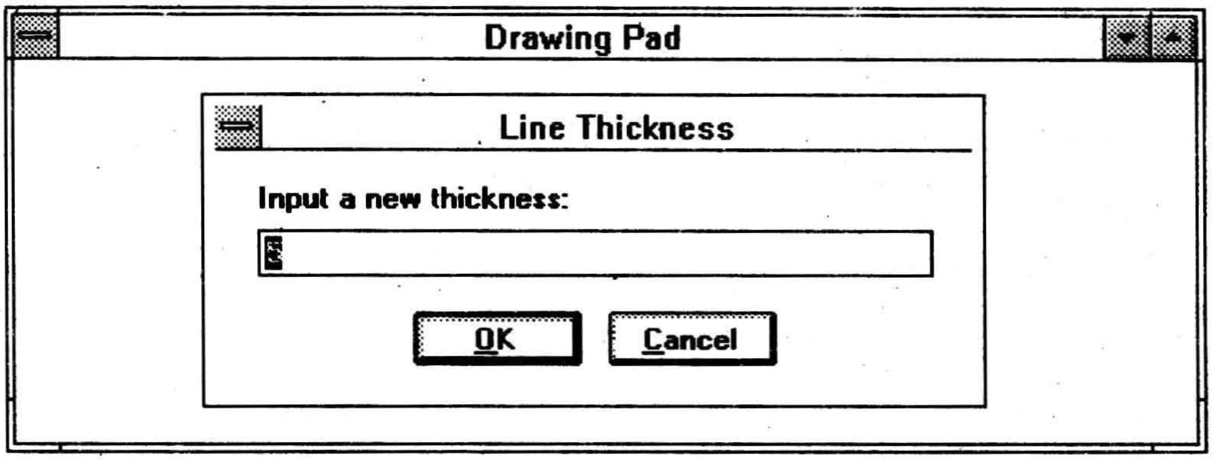
5.5.3 调用 SetPenSize 函数

通过调用 SetpenSize 来修改画笔尺寸,而不直接在 EvRButtonDown 函数中进行,尽管它是 TmyWindow 的成员函数,而且可以直接存取受保护的数据成员 Pen 和 Pensize。因为这样建立了一个公共存取函数来对数据进行修改,这在今后将变得很重要,尤其是当需要经常修改画笔时。

对于 TMyWindow 类,有一个公有成员函数 SetPenSize,该函数需要一个参数,即一个值为新的画笔宽度的整型参数。打开输入对话框后,需要处理用户输入,检查输入结果的有效性,而你所要做的仅仅是调用 SetPenSize,其他工作均由 SetPenSize 函数完成。

SetPenSize 是一个非常简单的函数。为了重新设定笔的尺寸，首先应当删除当前存在的画笔，然后设定 PenSize 为新的画笔尺寸，最后建立一个基于新尺寸的画笔。该函数的具体实现如下：

```
void TyWindow::SetPenSize(int newSize)
{
    delete Pen;
    PenSize=new Size;
    Pen=new TPen(TColor::BLack,PenSize);
}
```



5.5.4 清除画笔

由于 Pen 是一个指向 TPen 类对象的指针，而不是实际的 TPen 类对象，所以它不会在 TMyWindow 类对象析构的时候自动析构。故此，必须在 TMyWindow 类对象的析构函数中显式析构 Pen 对象。为达到此目的，可通过 delete 来实现。

TMyWindow 类结构的定义如下：

```
class TMyWindow:publicTWindow{
    public:TMyWindow(TWindow *parent=0);
        ~TMywindow() { deleteDragDC;delete Pen; }

        void SetPenSize(int newSize);
    Protected:
        TDC * Dra6DC;
        int PenSize;
        TPen * Pen;

        // Override member function of TWindow
        BOOL CanClose();

        // Message response functions
        void EvLButtonDown(UINT,TPoint&);
        void EvRButtonDown(UINT,TPoint&);
        void EvMouseMove(UINT,TPoint&);
```

```
    void EvLBUttonUp(UINT,TPoint&);
    DECLARE_RESPONSE_TABLE(TMyWindow);
};
```

5.6 第六步:重绘窗口和增加菜单

上一节介绍的程序还存在着几个缺陷,其中最主要的是,绘制的窗口不知道如何绘制自己。例如在窗口中画一条线,然后将该窗口最小化,最后再把最小化的窗口恢复至原来大小,你会发现,原来在窗口中画的线消失了。

另一个缺陷是用户只能通过鼠标来使用该应用程序。用户或者按下左键来划一条线,或者是按下右键来改变画笔尺寸。

在本节当中,你将会学到如何使窗口对其中的内容保持记忆以便在需要的时候对其进行重绘;另外你还可以增加一个菜单,以使用户有更多的方法使用该应用程序。

5.6.1 窗口重绘

在进行窗口重绘之前有两个问题需要解决:

1. 保存窗口内显示信息的方法。
2. 把保存的信息重新显示到窗口中去的方法。

一、图形保存

在前几节的应用程序中,用户按下鼠标左键并移动鼠标在窗口中画线。这种方法画线很方便,但并不能存储画线的点以备后用。

由于所画的线是由窗口的一系列点组成,你可以利用 ObjectWindows 的 TPoint 类来保存储画线点。由于每条直线都是由多个点组成的,所以你需要一个基于 TPoint 类的对象数组来存储一条直线。为了不做分配、空间管理、更新 TPoint 类对象数组等工作本节的应用程序将利用 Borland 公司的容器类 TArray 来定义一个新的数据类型 TPoints;用容器类 TArrayIterator 定义 TPointIterator。上述两种类型的定义如下所示:

```
typedef TArray<TPoint>TPoints;
typedef TArrayIterator<TPointsIterator;
```

TMyWindow 类加入一个 TPoints 对象用以存储直线上的点。实际上,它用一个指向 TPoints 对象的指针,一个具有保护属性的类成员,叫作 Line。它指向一个在构造函数中生成的 TPoints 类数组。构造函数如下所示:

```
TMyWindow::TmyWindow(TWindow * parent)
{
    Init(parent,0,0);
    DragDC=0;
    PenSize=1;
    Pen=new TPen(TColor::Black,PenSize);
```

```
    Line=new TPoints(10,0,10);
}
```

二、TPoints 类：

Borland C++容器类库、TArray 类及 TArrayIterator 类在《Borland C++程序员指南》一书的第七章有十分详细的讲解。这里只简单介绍一下如何在应用程序中使用这两个类。为了使用这两个类，在你的源文件中必须包含 Classlib 目录下的头文件 array. h。

类 TArray 的构造函数需要三个整型参数：

① 第一个参数代表数组的上界，即数组下标的最大值。

② 第二个参数代表数组的下界，即数组下标的起始值。该参数的缺省值为0，这是为了与C及C++的规则保持一致，在C及C++中，数组下标的缺省值为0。

③ 第三个参数代表数组的增长量。它指的是随着对象成员个数的增加，已定义的数组空间不足容纳下所有成员时，数组空间每次动态增加的量。

下面是在 TMyWindow 构造函数中为数组分配初始空间的语句：

```
Line=new TPoints(10,0,0);
```

上面生成的数组具有能容纳10个数组成员的空间，从0开始。一旦该数组存储了10个成员，而又试图在另外一个对象成员时，该数组能自动再增加能够容纳10个对象成员的空间。数组的这一动态增长机制可以使你从一个很小的保守数字开始，同时也解决一些伴随着静态数组的使用经常出现的问题，如空间耗尽、需重新分配内存和增加数组空间等。

一旦已经生成了一个数组，你需要能够使用它。TArray 类提供了若干函数可供使用，如增加成员，删除成员，清除数组及一些类似的函数。这里的应用程序仅仅使用了其中很少的几个函数。下面简单地描述这几个函数：

1. Add 函数向数组中增加一个成员。它需要一个参数，即对数组对象的一个引用。例如，欲增加一个 Tpoint 类对象到 TPoints 数组，应按如下方式进行：

```
// construct a TPoints array(an array of Tpoints objects)
    TPoints points(10,0,10);
// construct a TPoint object
    TPoint P(3,4)
// Add the TPoint object p to the array
    Points. Add(p);
```

2. Flush 函数清除数组中存储的所有成员，同时设置数组的尺寸为初始值。该函数不需要参数。为了清除上面代码生成的数组，应按如下方式调用 Flush 函数：

```
// Clear all the members in the points array points. Flush();
```

3. GetItemsInContainer 函数返回容器中项目的总数。注意这里的数目指的实际加到容器中的对象数目，而非可用容量数。例如，容器即使还能容纳30个对象，而目前只存有23个对象，在这种情况下，GetItemsIncontainer 函数的返回值为23。

三、TPointsIterator 类

包容类——这里为 TPointsInterator 类型——借助于它的帮助，可以在数组中移动，每次存取一个单独对象。包容类的构造函数需要一个参数，即一个对 TArray 类对象的引用(数组中对象的类型是通过包容类的定义建立的)。下面是当应用 TMyWindow 类的成员 Line 时，包容类的定义形式：

```
TPoinsIterator i( * line);
```

由于包容类的参数是一个对 TPoints 类对象的引用，而 Line 是一个指向 TPoints 类对象的指针，故只有用 * line 作为参数才能符合包容类构造函数对参数类型的要求。

建立了包容类，就可以借助它存取对象数组中的一个单独对象，从第一个开始，每次一个。在本章的应用程序中，包容类用得不是很多，这里不做过多的讲解，仅介绍所用到的关于包容类的两个性质。

■ 通过对包容类施加++操作符。可以遍历整个数组。每次返回一个对数组中当前对象的引用，并自动使包容类指向下一个对象。这两个动作的执行顺序取决于++操作是用包容类的前缀还是后缀操作符。如果用作前缀操作符，则先增加包容类的值使其指向下一个对象，然后返回对其的引用；若作为前缀操作符，则返回对当前所指对象的引用，然后增加包容类的值使其指向下一个对象。

当包容类移出数组空间时，自动回到零初始位置。这可利用下面的程序测试：

```
TPointsIterator i( * Line);
while(i)
    i++;
```

■ 可以利用函数 Current()得到包容类当前所指的对象。它返回的亦是对该对象的引用。在下面的程序中，首先利用该函数得到一个对象，然后与另一个对象进行比较：

```
TPointsIterator i( * line);
TPoint tmp(5,6,);
if(i.Current()==tmp)
    return TRUE;
else
    return FALSE;
```

四、应用数组类

在 TMyWindow 类的构造函数中创建了一个 Line 数组，在如下所述的四个地方对其进行存取：

■ 在 EvLButtonDown 函数内部。在函数开始部分先对 Line 数组刷新，然后清屏。直线初始点在函数返回前加入到数组中。该函数代码如下：

```
void TMyWindow::EvLButtonDown(UINT,TPoint& point)
    {
```

```
        Line-> Flush();
        Invalidate();
        if(!DragDC){
            SetCapture();
            DragDC=new TClientDC(*this);
            DragDC->SelectObject(*pen);
            DragDC->MoveTo(point);
            Line->Add(point);
        }
    }
```

■ EvMouseMove 函数，用户移动鼠标在窗口内画线时，画线的每个点都被存到 Line 数组中。EvMouseMove 函数的代码如下：

```
void TMyWindow:EvMouseMove(UINT,TPoint& point)
    {
        if(DragDC)
        {
            DragDC->LineTo(point);
            Line ->Add(point);
        }
    }
```

■ Paint 函数。该函数的功能及源程序代码在稍后一点讲解。

■ CmFileNew 函数。该函数将在第七节"应用通用对话框"中介绍。

五、Paint 函数

在标准 C Windows 程序中，如果你打算手工重绘窗口，你可以通过截获 WM_PAINT 消息来做需要做的工作以实现窗口重绘。这也许会使你想到，窗口重绘的正确方法应该是在 TMyWindows 类的消息响应表中增加宏 EV_WM_PAINT，同时建立一个叫作 EvPaint 的函数。

你可以按照上面的方法来进行，这安全取决于你的意愿。但更好的方法应该是重载 TWindow 类的成员函数 Paint。TmyWindow 的基类 TWindow 在其成员函数 EvPaint 中确实做了许多工作，如进行了 BeginPaint 和 EndPaint 调用，为窗口生成了设备上下文等。

Paint 是 TWindow 类的一个虚函数。TWindow 类的成员函数 EvPaint 对它进行调用，缺省的 Paint 函数什么也不做，为一空函数。你可以通过它提供从 TPoints 数组画线所需要做的一些特殊的处理过程。

下面是 Paint 函数的原型，它需要加到 TMyWindow 类中。

```
void paint(TDC&,BOOL,TRect&);
```

其中：

① 第一个参数为调用函数建立的设备上下文；你进行窗口重绘的时候会用到它。

② 如果第二个参数为 TRUE，则在窗口重绘之前要清除屏幕上原有信息。如果为

FALSE，则在重绘之前不清除屏幕窗口内的信息。

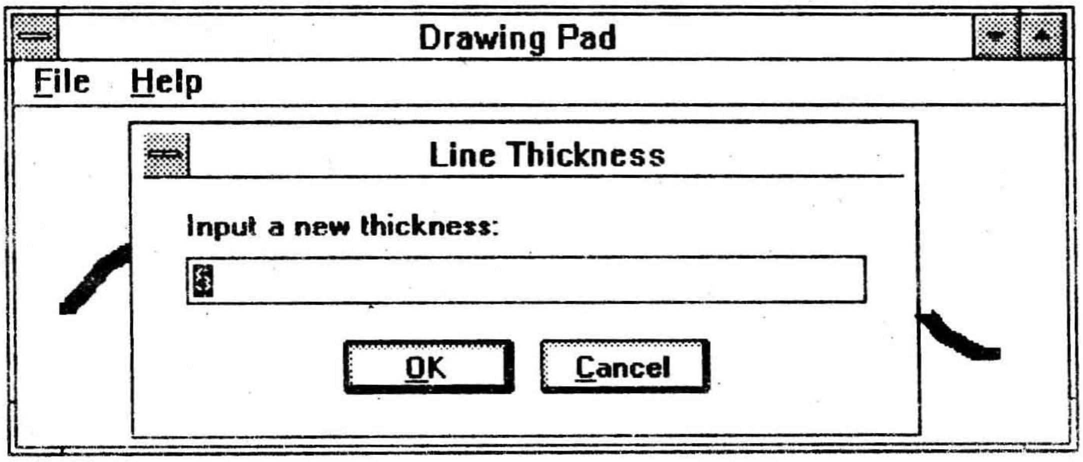
③ 第三个参数指示设备上下文中需要重绘的区域。

在当前情况下，假设你总是想清除屏幕窗口内的信息，而且整个窗口区域都需要重绘。Paint 函数实现如下基本算法：

- 创建一个包容类遍历直线上的点；
- 把画笔选入设备上下文，该设备上下文由调用函数传来。
- 如果遍历的是数组中的第一个对象，则将当前坐标设为该对象包含的值。
- 如果数组中还有其他的点，则从当前点画线至某一点，该点的位置由当前数组成员包含的值决定。

为此，TMyWindow 的 Paint 函数应定义为：

```
void TMyWindow::Paint(TDC& dc,BOOL,TRect&)
{
    BOOL first=TRUE;
    TPointsIterator i(*Line);
    dc.SelectObject(*pen);
    while(i) {
        TPoint p=i++;
        if(!first)
            dc.LineTo(p);
        else{
            dc.MoveTo(p);
            first=FALSE;
        }
    }
}
```



5.6.2 菜单命令

为了在你的应用程序中增加一个菜单以及相应的事件处理过程，还有许多工作有待于

完成：

■ 为菜单选项定义事件标识符。一般来讲，标示符都是由大写字母组成，并且以 CM_ 开头，例如，菜单选项"File OPen"的标识符为 CM_FILEOPEN。

■ 在你的资源文件中加入适当的菜单资源。

■ 在类中加响应菜单选项的事件处理函数。在 ObjectWindows 2.0中，习惯将事件处理函数与事件标识符同名，只是省略了下划线，取首字母为大写，其余为小写字母。例如，事件 CM_FILEOPEN 的处理函数名为 CmFileOpen.

■ 在你的类的响应表中增加宏 EV_COMMAND，使事件标识符与事件处理函数建立对应关系。这个宏需要两个参数；第一个是事件标识符，第二个是事件处理函数。例如，"File Open"菜单选项的响应表入口为：

```
EV_COMMAND(CM_FILEOPEN,CmFileOpen)
```

■ EV_COMMAND 宏要求事件处理函数不能有返回值，而且不应该有参数。所以"File Open"菜单选项的事件处理函数的原型就为：

```
void CmFileOpen();
```

一、加入事件标识符

前面所述的各个菜单选项都应赋予一个事件标识符，其定义如下所示：

```
#define  CM_FILENEW      201
#define  CM_FILEOPEN     202
#define  CM_FILESAVE     203
#define  CM_FILESAVEAS 204
#define  CM—ABOUT        205
```

这些标识符的定义放在资源描述文件 STEP06.RC 里，而不是放到一个单独的头文件里。从而可以减少工程文件中文件的数目，易于维护，同时还利于保持标识符的值在资源文件及应用源文件中相一致。

在资源文件中，具体的资源定义被放在一个由#ifndef 和#endif 定义的块中。如下所示：

```
#ifndef RC_INVOKED
    // Resource definition here
    ⋮
#endif
```

RC_INVOKED 可以被所有的资源编译器所接受，但不能被 C++编译器所承认。标识定义应放在#ifndef/#endif 块之外，通常放在文件的首部。

二、加入菜单资源

现在，本节的应用程序将加入如下五个菜单选择项：

■ File New

- File Open
- File Save
- File Save As
- About

每个菜单选项都应与正确的事件标识相联系；例如，"File Open"选项应与CM_FILEOPEN相关联。

菜单资源是通过InitMainWindow函数加入到应用程序中去的，而且调用了AssignMenu函数。为了获取应用程序的主窗口，可通过调用GetMainWindow函数。因此可按如下方式定义InitMainWindow函数：

```
void InitMainWindow();
{
    SetMainWindow(newTFrameWIdow(0,"Drawing Pad",new TMyWindow));
    GetMainWindow()->AssignMenu("COMMANDS");
}
```

三、加入事件响应表项

为了把每个事件标识与相应的事件处理过程相关联，把下列的语句行加到事件响应表中：

```
EV_COMMAND(CM_FILENEW,CmFileNew),
EV_COMMAND(CM_FILEOPEN,CmFileOpen),
EV_COMMAND(CM_FILESAVE,CmFileSave),
EV_COMMAND(CM_FILESAVEAS,CmFileSaveAs),
EV_COMMAND(CM_ABOUT,CmAbout),
```

四、加入事件处理过程

现在需要为刚刚加入到相应表中的每个事件提供一个处理函数。由于这些处理函数有可能会变得很冗长，因此最好在类中声明函数，而在类外定义函数。

这里所涉及到的函数声明为：

```
voidCmFIleNew();
void CmFileOpen();
void CmFileSave();
void CmFileSaveAs();
void CmAbout();
```

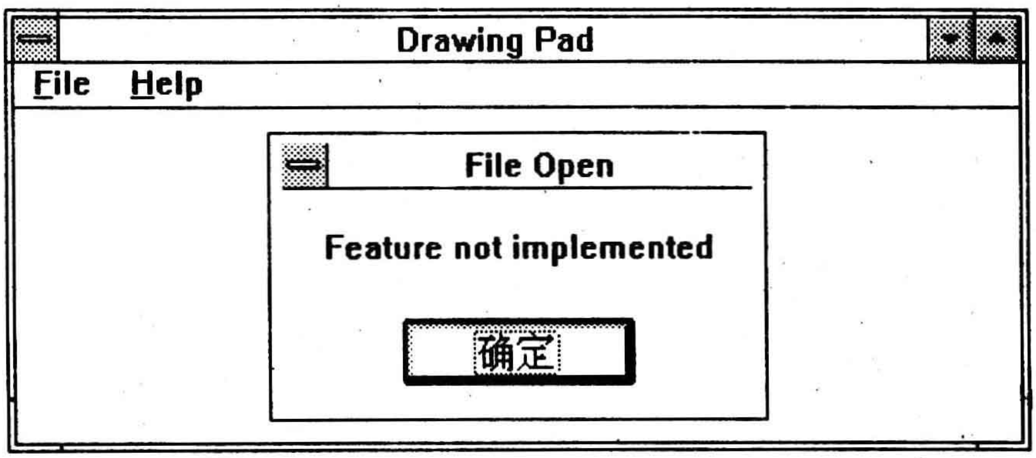
五、完成事件处理过程

完成事件处理的最后一步是定义函数。为了简单起见，使上面定义的函数完成尽可能少的功能。大部分函数仅仅弹出一个消息框，提示用户指定功能尚未实现。这样运行的函数有CmFileOpen，CmFIleSave，CmFileSaveAs和CmAbout。例如CmFileOpen()的定义为：

```
void TMyWindow::CmFileOpen()
{
    MessageBox("Feature not implemaented","File
            open",MB_OK);
}
```

CmFileNew 的功能比较简单，易于实现。所需要做的工作仅是清除已存储的原有画线点和清除屏幕窗口内的区域：

```
void TMyWindow::CmFileNew()
{
    Line->Flush();
    Invalidate();
}
```



5.7 第七步：应用通用对话框

在上一节里，应用程序中增加了几个事件处理函数，如 CmFIleOpen，CmFileSave，CmFileSaveAs 等，这两个函数的具体功能将在本节中得到实现。在实现的过程中，都用到了 ObjectWindows 的 TFileOpenDialog 类与 TFileSaveDialog 类。这两个类封装了 Windows 的 Open 和 Save 通用对话框以提示用户输入文件名。

利用 CanClose 函数可检测出在放弃当前所绘图形时，所绘图形与上次存盘相比是否做了修改。如果有所改动，用户将有机会确定是否存盘、或不存盘继续进行，或者结束整个操作。

为了实现 CmFIleOpen，CmFIleSave，CmFileSaveAs 函数的功能，还需要为前面定义的窗口类提供两个保护的(protected)成员函数：OpenFile 和 SaveFile。这两个函数将会在本节稍后一点讲解。

5.7.1 修改 TMyWindow 类

为了对菜单命令提供支持，需要对 TMyWindow 类增加三个新的数据成员：FileData，Isdirty，IsNewFile。

(一)FileData 成员

FileData 是一个指向 TOpenSaveDialog::TData 对象的指针。TOpenSaveDialog 类是 TFileOpenDialog 和 TFileSaveDialog 的基类。这两个派生类都用到了 TOpenSaveDialog::TData 类来保存与当前文件有关的信息。例如文件名、寻找文件的初始目录,文件名过滤器等。

由于在 TMyWindow 类的构造函数中,FileData 被初始化一个指向 TOpenSaveDialot::TData 类对象的指针,因此在 TMyWindow 的析构函数中,必须增加一个 delete 语句将该对象从内存中删除。

(二)IsDirty 成员

成员 IsDirty 是一个标志量,用来指明当前所绘图形是否"脏",意思是指所绘图形在做最后修改以来是否已经存盘。如果所绘图形尚未修改,或者根本未绘制任何图形,IsDirty 的值设定为 FALSE,否则,IsDirty 的值为 TRUE。在 TMyWindow 类的构造函数中,IsDirty 的值设定为 FALSE,这是因为还尚未绘制任何图形。

在 TMyWindow 类的构造函数之外,还有如下几个函数要对 IsDirty 赋值:

- 在 EvLButtonDown 函数中,IsDirty 赋值为 TRUE,用以表明对图形的修改。
- 在 CmFileNew 函数中,在窗口被清除后,IsDirty 赋值为 FALSE。
- 在 OpenFile 和 SaveFile 函数内部,IsDirty 被赋值为 FALSE,表示图形在存盘以后还没有被修改。

(三)IsNewFile 成员

根据 IsNewFile 可以判定文件是否已经有了一个名字。如果窗口内的图形是从一个已存在的文件中调入的,或者已经存储到了硬盘上的某个文件中,则表明文件已经有了名字,此时,IsNewFile 被赋值为 FALSE。在 TMyWindow 类的构造函数中,IsNewFile 被赋值为 TRUE,因为此还没有存过盘。

在 TMyWindow 类的构造函数之外,还有两个函数需要改变 IsNewFile 的值:

- 在 CmFileNew 函数内部,清除窗口之后,IsNewFile 的赋值为 TRUE。
- 在 OpenFile 和 SaveFile 函数内部,IsNewFile 赋值为 FALSE,表示文件已经存过盘。

5.7.2 增强 CanClose 函数的功能

在前几节中应用程序里的 CanClose 函数还存在着几点不足:首先,每次被调用,它都会提示用户存储所绘图形,如果图形尚未被修改,这完全没有必要。其次,仅仅一个"Yes"和一个"No"选择项还不能满足要求,例如,若用户不想关闭窗口,此时期望的响应是"CanClose",即不是"Yes",也不是"No",因此应该有三个响应选择,如下图所示:

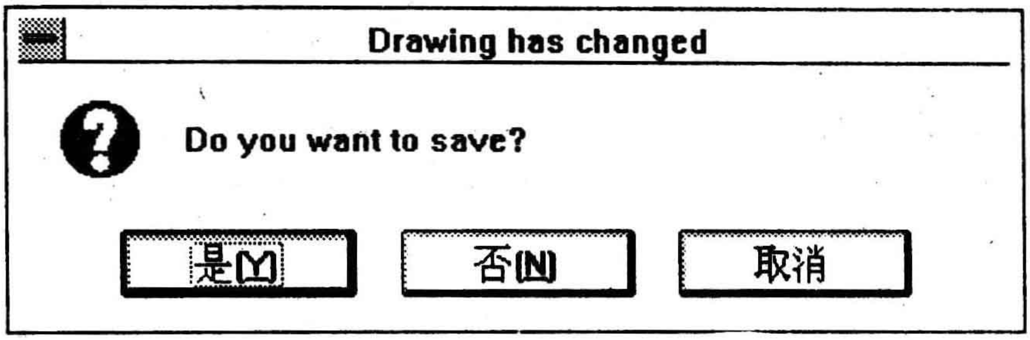


图5.6 CanClose 函数运行时弹出的对话框

CanClose 函数通过检测 IsDirty 的值来确定是否有必要提示用户是否真的要结束操作。如果所绘图形尚未改动,则没有必要这样做。用户可直接装载新的文件。

如果图形有改动,CanClose 函数则弹出一个对话框,并在框中给出三个可能的选择:

- 选择"取消"(cancel)按钮,表明用户要退出关闭操作。此时,消息对话框返回值是 IDCANCEL,CanClose 函数的返回值是 FALSE。
- 选择"是"(yes)按钮,表明用户希望对文件进行存盘,此时对话框的返回值是 IDYES,CanClose 函数的返回值为 TRUE。CanClose 函数在结束之前,将调用 CmFileSave 函数完成期望的工作。
- 选择"否"(No)按钮,表明用户不想对文件进行存盘。此时,CanClose 函数不做什么工作,直接返回 TRUE。

Canclose 函数的代码为:

```
BOOL TMyWindow:CanClose()
{
    if(IsDirty)
        switch(MessageBox("Do you want to save?","Drawing has changed",
                          MB_YESNOCANCEL | MB_ICONQUESTION)){
          case IDCANCEL:
                // Choosing Cancel means to abort the close -- return FALSE.
                return FALSE;

          case IDYES:
              // Choosing Yes means to save the drawing.
              CmFileSave();
        }
    return TRUE;
}
```

5.7.3 CmFileSave 函数

CmFileSave 函数相对而言比较简单,它通过检测 IsNewFiloe 的值来确当前所绘图形是否为一否新图,若是,IsNewFile 的值 TRUE,CmFileSave 函数若调用 CmFileSaveAs 函数,提

示用户输入一文件名。否则,直接调用 SaveFile 函数完成图形存储。

CmFileSave 函数的代码为:

```
void TMyWindow:CmFileSave()
{
    if(IsNewFile)
        CmFileSaveAs();
    else
        SaveFile();
}
```

5.7.4 CmFileOpen 函数

CmFileOpen 函数也非常简单。首先,它调用 CanClose 函数检测是否可以关闭当前所绘制的图形,若 CanClose 返回值为 FALSE,则 CmFileOpen 直接返回。

如果 CanClose 的返回值是 TRUE,则 CmFileOpen 建立一个 TFileOpenDialog 类对象。TFileOpenDialog 类的构造函数最多可接受五个具体参数,但本例中只用两个参数,其余三个参数均取缺省值。在这两个需要提供的参数中,一个是指父窗口的指针,另一个是对 TOpenSaveDialog::TData 类对象的引用。这里,指向父窗口的指针为 this;TOpenSaveDialog::TData 类对象由 FileData 提供。

一旦成功创建了对话框,便可调用 TFileOpenDialog 类的成员函数 Execute 来执行它。该对话框仅有两个返回值,LOOK 和 IDCANCEL。如果用户按下的"OK"按钮,则返回值为 IDOK,若按下的是"Cancel"按钮,则返回值为 IDCANCEL。

如果返回值是 IDOK,则 CmFileOpen 调用 OpenFile 函数来打开具体的文件。Execute 函数同时将文件名存储到 FileData 对象的 FileName 成员里。如果返回值不是 IDOK,而是 IDCANCEL,则 CmFileOpen 不做更多的工作,直接返回。

CmFileOpen 函数的代码如下:

```
void TMyWindow::CmFileOpen()
{
    if(canClose())
    if(TFileOpenDialog(this, * FileDate).Execute()==IDOK)
      OpenFile();
}
```

文件打开对话框的屏幕显示:

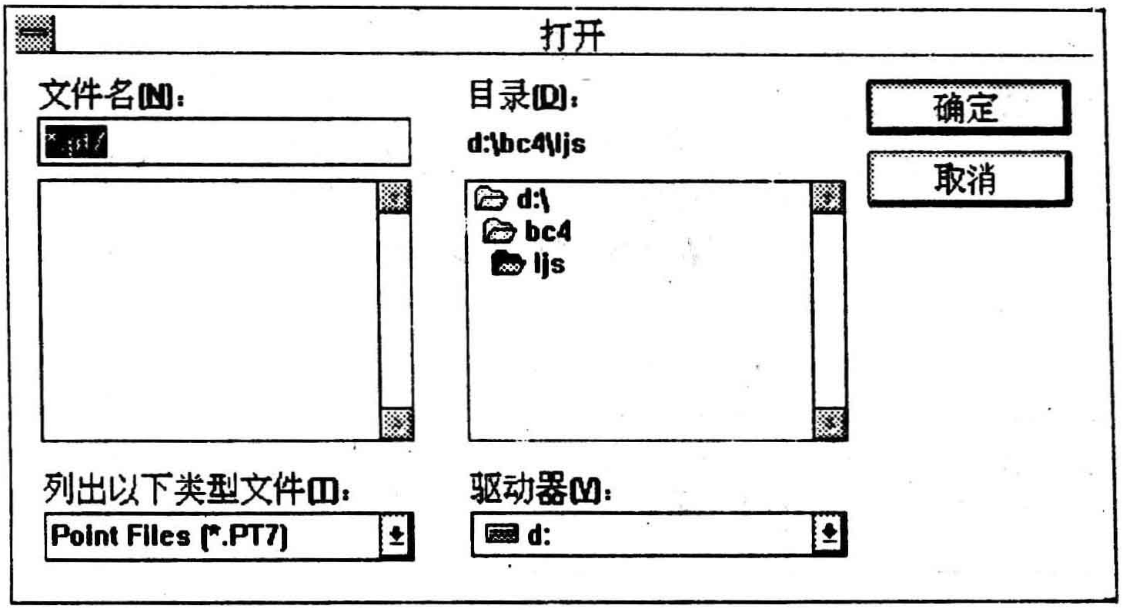


图5.7 打开文件时弹出的对话框

5.7.5 CmFileSaveAs 函数

CmFileSaveAs 函数有两个功能：一个是用一个新文件名存储一幅新图；另一个是可以换名存储一幅已存在的图形。

CmFileSaveAs 通过检查 IsNewFile 的值来决定执行哪一个功能。如果是一个新文件，它拷贝一个空的字符串到 FileData 的 FileName 成员。否则，FileName 的内容不变。

这两种情况的区别是至关重要的。如果 FileName 存储的是一个空串，则在文件保存对话框中的文件名区域里，缺省的文件名被设定为文件名过滤器的值，这里为"*.pts"。但是如果 FileName 已经存储了一个名字，则该名字及其路径一起被显示在对话中的文件名区域里。

至此，文件存储对话框已经创建并执行。其运行方式与 CmFileOpen 函数中的文件打开对话框非常一致。如果 Execute 函数返回 IDOK，CmFileSaveAs 函数将调用 SaveFile 函数。

CmFileSaveAs 函数的代码如下：

```
void TMyWind::CmFileSaveAs()
{
    if(IsNewFile)
        strcpy(FileData->FileName,"");
    if(new TFileSaveDialog(this, * FileData)->Execute==IDOK)
        SaveFile();
}
```

执行后显示的对话框如下：

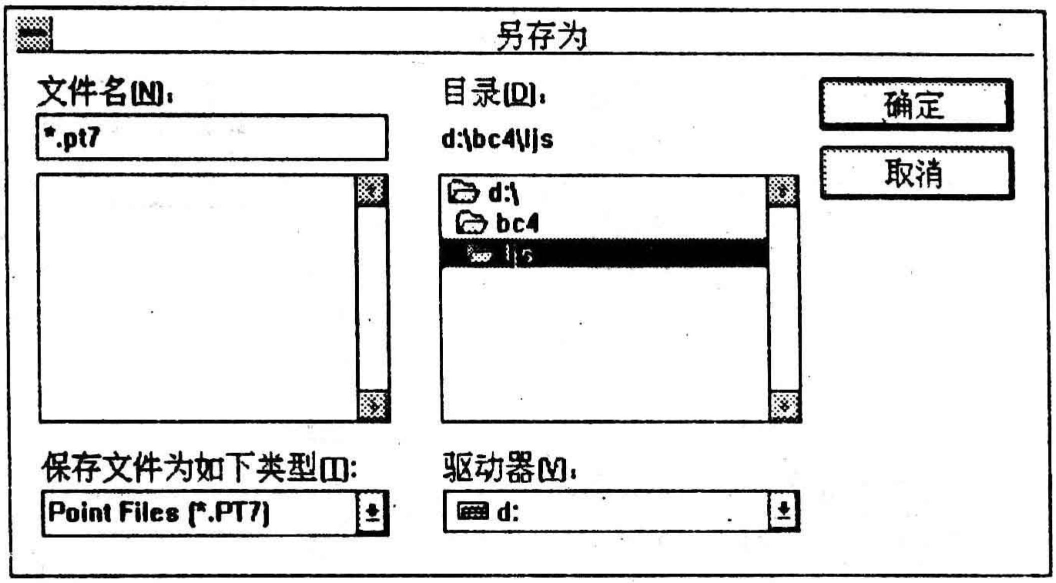


图5.8 存文时弹出的对话框

5.7.6 打开与存储图形文件

用户可以通过 CmFileOpen,CmFileSave 和 CmFileSaveAs 函数来存取图形文件。但实际的工作是由 OpenFile 和 SaveFile 函数来完成的。下面将描述一下这两个函数是如何工作的,但不对整个程序做技术上的详细讲解。

(一)**OpenFile** 函数

OpenFile 函数以 ifstream 方式打开由 FileName 命名的文件,ifstream 是标准的 C++输入/输出流。如果指定的文件由于某种原因不能打开,OpenFile 函数将弹出一个消息,通知用户该文件不能打开,然后返回。

一旦文件打开成功,Line 数组将被刷新。OpenFile 首先读入存储的数据总数,即文件中的第一个元素。然后建立一 for 循环,读进后面存储的点,并加到 Line 数组中。

读完所有的数组点,OpenFile 调用 Invalidate 函数清除屏幕区域,并发送 WM_PAINT 消息,一幅新的图形便显示在窗口内。

最后 OpenFile 给 IsDirty 和 IsNewFile 均赋值 FALSE。

该函数的代码如下:

```
void TMyWindow::OpenFile()
{
    ifstream is(FileData->FileName);

    if(!is)
        MessageBox("Unable to open file","File Error",
                   MB_OK | MB_ICONEXCLAMATION);

    else{
        Line->Flush();
        unsigned numPoints;
```

```
        is >>numPoints;
        while (numPoints--) {
            TPoint point;
            is >> point;
            Line->Add(point);
        }
    }
    IsNewFile=IsDirty=FALSE;
    Invalidate();
}
```

(二)SaveFile 函数

SaveFile 函数以标准输出流的方式打开由 FileName 命名的文件。若由于某种原因不能成功打开指定文件,则弹出一个消息框,提示用户文件打开失败,然后返回。一旦文件打开成功,对象 Line 的成员函数 GetItemsInContainer 被调用,并将返回的结果存入文件,以供 OpenFile 读取以确定文件存储的点数。然后,建立一个基于 Line 的包容类 i,i 检测出 Line 数组中的所有点并将其存入刚打开的文件里。

最后,IsNewFile 和 IsDirty 均被赋值 FALSE,下面是该函数的代码:

```
void TMyWindow::SaveFile()
{
    ofstream os(FileData->FileName);
    if(! os)
        MessageBox("Unable to open file","File Error",
                    MB_OK|MB_ICONEXCLAMATION);
    else
    {
        os <<LIne->GetItemsInContainer();
        TPointsIterator i( * Line);
        while(i)
            os<<i++;
        IsNewFile=Isdirty=FALSE;
    }
}
```

5.7.7 CmAbout 函数

CmAbout 函数演示如何使用 ObjectWindows 中的自定义对话框,其使用方法异常简单,该函数仅有一行代码。通过应用 TDialog 类与 IDD_ABOUT 对话框资源在屏幕上弹出信息对话框。如下图所示:

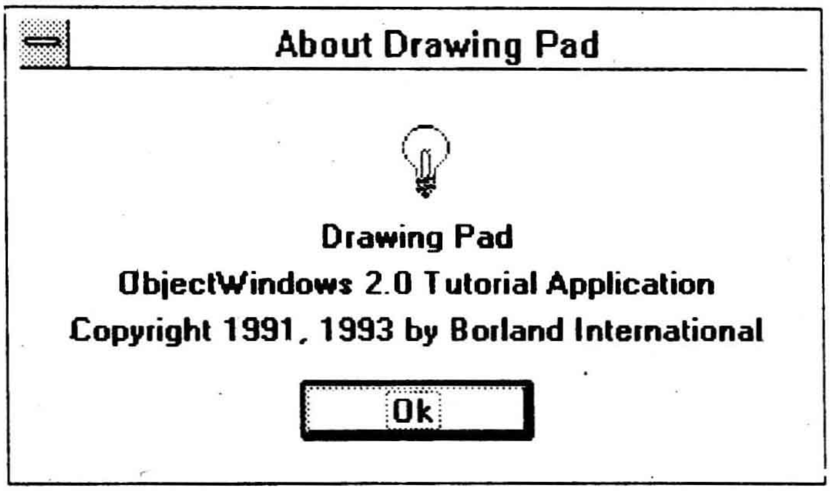


图5.9 选择 About 时弹出的对话框

TDialog 类对象的创建需要三个参数：

- 第一个参数是指向父窗口的指针。与 TFileOpenDialog 和 TFileSaveDialog 类的构造函数一样，可以使用 this 指针，设定父窗口为 TMyWindow 类的对象。
- 第二个参数是对 TResID 类对象的引用。它应当是所用对话框资源的标识。
  通常，没有必要向其传递一个对 TResID 对象的引用，就像操纵直接由 Windows API 函数创建的对话框一样，向其传递一个资源标识或一个字符串，TResID 类中的类型转换操作符自动将其转换为正确的类型。
- 第三个为指向 TModule 类对象的的指针：TModule *，通常取缺省值。

对话框对象创建完成以后，剩下的工作就是调用 Execute 函数，当用户关闭对话框，运行结束，CmAbout 函数返回。刚刚创建的对话框对象自动消失。

CmAbout 函数的代码如下：

```
void TMyWindow::CmAbout()
{
    TDialog (this,ID_ABOUT).Execute();
}
```

5.8 第八步：在窗口中画多条线

从功能上讲，本节的应用程序有很大改进。程序中又增加了一个新类，TLine，它是由 TPoints 类派生出来的。基于 TLine 类，派生一个包含多个 TLine 对象的类 TLines。有了这些类的帮助，很容易在窗口中画多条线。另外，为了图形存储方便，可以加入一个流操作符。最后需要开发新的 paint 函数，处理在窗口中画多条线。

5.8.1 TLine 类

类 TLine 是从 TPoints 类公有派生出来的，因此它继承了基类 TPoints 的所有功能。其

中包括前面已经用过的函数 Add,Flush 和 GetItemsIncontainer。另外,还可以按前面所用的方式继续将 TPointsIterator 作用类 TLine。

由于增加了新类,故可以增加一些附加的功能。例如可以增加一个数据成员保存画每条线时笔的尺寸。为了避免直接存取该数据成员,增加一个存取函数来完成相应的操作。

在 TLine 类中,有一个保护类型(protected)的数据成员 PenSize 用来保存笔的尺寸。PenSize 由同名的两个函数来访问。这两个函数均返回包含画笔尺寸的整型(int)数。下面介绍这两个函数的区别:

- 第一个 QueryPen 函数没有参数,返回笔的尺寸。
- 第二 QueryPen 第一个参数,int 类型。该函数把 PenSize 设定为参数传入的值,并返回 PenSize 的返回值,得知其是否把笔设定为输入的尺寸;该函数也对参数传入的值进行检查,确保它是合法的尺寸。

TLine 类定义了一个"=="操作符,用以检查两个对象是否相同。若相同则返回 TRUE。定义"=="操作符对定义 TArray 类的数组是必需的,因为 TArray 类的对象应该有该操作符。

最后,还需要声明两个操作符"<<"和">>",这两个操作符是 TLine 类的友元。其具体功能将在后面实现,它们为 SaveFile 和 OpenFile 函数中的流操作提供了诸多方便。

下面是 TLine 类的声明:

```
class TLine:public TPoints{
    public:
        TLine(int penSize=1):TPoints(10,0,10){PenSize=penSize;}

        int QueryPen() const { return PenSize;}
        int QueryPen(int penSize);

        // The == operator must be defined for the container class,even if unused
        BOOL operator==(const TLine& other) const { return &other==this;}
        friend ostream& operator <<(ostream& os,const TLine& line);
        friend istream& operator >>(istream& is ,TLine& line);
    protected:

        int PenSize;
};
```

5.8.2 TLines 类数组

上面定义了 TLine 类,现在可以由此定义 TLines 和 TLinesIterator 类数组。这两个数组的工作方式与前面定义的 TPoints,TPointsIterator 完全一致,仅是后者存储的是 TPoints 类对象,而前者存储的是 TLines 类对象。

下面是 TLines 和 TLines Iterator 的定义:

```
typedef TArray<TLine> TLines;
typedef TArrayIterator <TLine> TLinesIterator;
```

5.8.1 TLine 对象的输入与输出

通常，大多数对象需要存储到文件中，或者从文件中读出，这两种操作分别是借助于操作"<<"和">>"完成的。由于把"<<"和">>"以使它们能处理 TLine 封装的特殊类型的数据。

定义了这两个操作符，仅用一行源代码就可将一个完整的 TLine 对象存入文件。下面将介绍在改变了.OpenFile 和 SaveFile 函数之后，该功能是如何实现的。

(一)"<<"操作符

大体上讲，"<<"具有前面内容中介绍中介绍的 SaveFile 函数的功能。它不需要打开一个文件(这是由使用该操作的函数完成的)，另外它还有一个附加的数据(penSize)需要输入。除此之外，它与第七节中的 SaveFile 没有什么区别。下面是<<的定义：

```
ostream& Operator <<(ostream& os,const TLine& line)
{
    // write the number of points in the line
    os<< line.GetItemsInContainer()<<'\n';
    // write the pen size
    OS<< ' '<<line.PenSize;
    // Get an iterator for the array of points
    TPointsIterator j (line);
    // While the iterator is valid (i.e. it has n't run outof points while (j)
    // write the point from the iterator and increment the array
        os <j++;
        os<<'\n';
    // return the stream object
        return os;}
```

(二)>>操作符

"》"操作的功能与 OpenFile 函数的功能基本一致，只不过是它不需要亲自打开文件，另外还需要多输入一个数据。下面是该操作符的定义：

```
istream& operator >>(istream & is ,TLine& line)
{
    unsgned numPoints;
    is >> namePoints;
    is>> line.PenSize;
    while(numPoints--)
    {
        TPoint point;
        is>> point;
        line.Add(point);
```

```
    }
    // return the stream object
    return is;
}
```

5.8.3 TMyWindow 类的扩充

为了适应新定义的 TLine 类的需要，TMyWindows 应做一系列改动，首先看一个数据成员的改动：

- PenSize 被取消，因为每条单独的线都包含着相应的画线尺寸。
- 数据成员 Line 由 TPoints * 变为 TLine * 类型，Line 对象保存着画当前线的点。
- 增加了一个数据成员 Lines，为 TLines * 类型，Line 对象保存着所有原 TLine 对象。

下面是有关成员函数的变化：

- SetPenSize 函数变为保护的成员函数，因为画笔、尺寸的变化要传给 TLine 类。SetPenSize 应在 TMyWindow 类的内部使用。同时 SetPenSize 函数通过调用 QueryPen 函数为当前画线设置画笔尺寸。
- 增加了 GetPenSize 函数，该函数执行了 TInputDialog 的功能，因为 EvRButtonDOWN 和 CmPenSize 函数都用到该种类型的对话框(TInputDialog)。
- EvRButonDown 函数调用 GetPenSize 来打开一个输入对话框。
- CmPenSize 函数处理 CM_PENSIZE 事件。为用户提供了一种新的改变画笔尺寸的方法。
- OpenFile 和 SaveFile 函数做了修改，用以存取一组 TLine 对象，而不是一组 TPoint 对象。由于使用了"<<"和">>"操作符，这两个函数从形式上基本上没有变化。

另外，Paint 函数有较大的变化，在下面讨论。

(一)**Paint** 函数

Paint 函数有两个包容类，而不是一个。因为它要从画线数组检测出每条线，对于每条线，不仅要能确定画线的宽度，而且还要能确定组成这条线的所有点。

为了实现上述功能，Paint 首先建立 Lines 的包容类，它从线数组中检查出每条线。对于每条线，Paint 首先查询笔的尺寸，然后设置 Window 笔的尺寸并把它选入设备上下文。最后为当前所画的线建立一个包容类，从中取出画线的点。

Paint 函数余下的部分与第七节中的一样，即根据得到点在窗口中画线。

下面是 Paint 函数的代码：

```
void TMyWindow::Paint(TDC& dc,BOOL,TRect &)
{
    // Iterates through the array of line objects
    TLinesIterator i( * Lines);
    while(i){
    // Set pen for the dc to current line's pen
```

```
        TPen pen (TColor::Black,i.Current.QueryPen());
        dc.SelectObject(pen);
        // Iterat through the line for points
        TPointsIterator j(i++);
        BOOL first=TRUE;
        while(j){
            TPoint p=j++;
            if(!first)
                dc.LineTo(p);
            else
                dc.MoveTo(p);
                first=FALSE;
           }
          }
      }
  }
```

5.9 第九步:修改画笔

在本节的应用程序中,为了使用户能够画具有不同尺寸和不同颜色的线,为 TLine 在增加了一个新的数据成员 TColor 和一个新的成员函数 CmPenColor。该函数处理 CM_PEN-COLOR 菜单命令,并通过应用 TChooseColor 弹出一个颜色选择框,如下图所示,供用户选择自己喜爱的颜色。除此之外,还为 TLine 类增加了几个辅助函数以便使其他类能方便地存取 TLine 封装的信息。

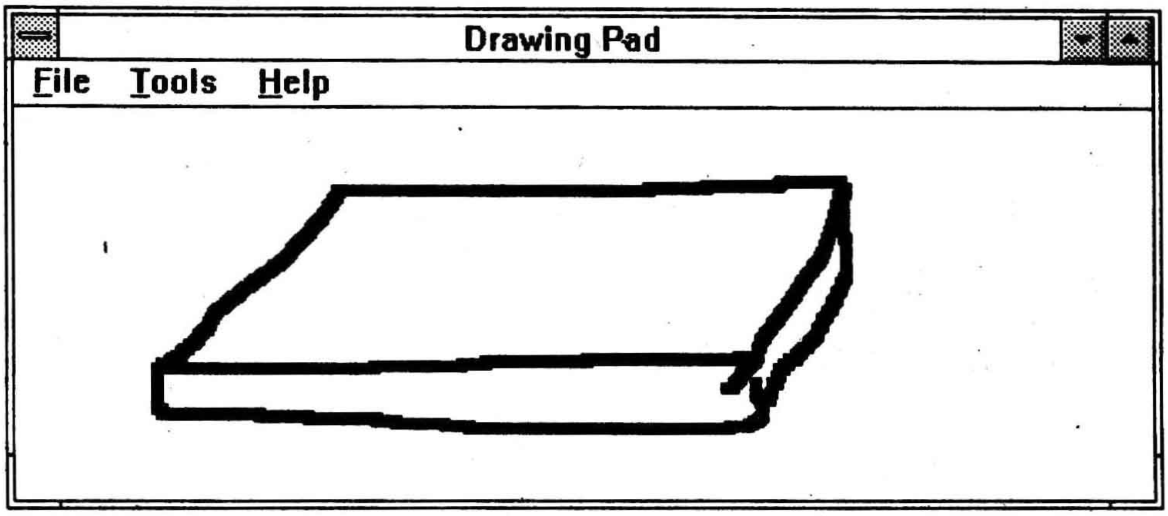


图5.10 颜色选择对话框

除了为笔增加了颜色功能,还增加了处理 TLine 类新添加属性的流操作符,以及为简化 Paint 函数而设计的 Draw 函数。

5.9.1 修改 TLine 类

为了实现新的功能，对 TLine 类的声明需要做如下改动：

- 新的保护数据成员。Color(一个 TColor 类对象)。Color 和 PenSize 共同组成了画笔的属性。
- 构造函数由

  ```
  TLine(int penSIze=1);
  ```

  变为：

  ```
  TLine(const TColor &color=(TColor)0,int penSize=1);
  ```

  它自动将 PenSize 赋给其第二个参数来创造一个 TPen 类对象，并将所创造的对象赋给 Pen。若没有指定参数，第一个参数则取缺省值：TColor::Black 作为笔的颜色。
- 删去了两个 QueryPen 函数，增加了如下三个新函数：返回画笔尺寸的 QueryPenSize 函数；返回画笔颜色的 QueryColor 函数和返回当前画笔的 QueryPen 函数。
- 增加了两个 SetPen 函数设置画笔属性。第一个函数仅带一个整型参数，第二个函数带三个参数，即一个对 TColor 类对象的引用与两个整型数。有关其详细情况将在后面讨论。
- 增加了一个虚函数(virtual)Draw，它可以使 TLine 类重绘原有图形。由于是虚函数，可以在派生类中对其重载。

下面是 TLine 类的声明：

```
class TLine:public TPoints{
    public:
        // constructor to allow construction from a color and a pen size.
        // Also serves as default constructor.
        TLine(const TColor &color=TColor(0),int penSize=1
          :TPoints(10,0,10),PenSize(penSize),Color(color){}

        // Functions to modify and query pen attributes.
        int QueryPenSize(){return PenSize;}
        TColor&QueryColor(){Return Color;}
        void SetPen(TColor &newColor,int penSize=0);
        void SetPen(int penSize);

        // TLine draws itself. Returns TRUE if everything went OK.
        virtual BOOL Draw(TDC &)const;

        // The==operator must be defined for the container class,even if unused
        BOOL operator==(const TLine& other)const {return & other==this;}
        friend ostream& operator <<(ostream& os,const TLine&,line);
        friend istream& operator >.(istream& is,TLine&,line);

    protected:
        int PenSize;
```

```
        TColor Color;
    };
```

(一)画笔存取函数

第八节中的 QueryPen 函数既能返回画笔的尺寸,又能设置画笔的尺寸。本节 TLine 中的 QueryPenSize 和 QueryColor 函数不能改变属性,仅能返回有关画笔属性的信息。

有两个 SetPen 函数可以设置画笔属性。第一个仅能设置画笔尺寸,第二个可以设置画笔的颜色、尺寸、画笔的类型。利用第二个 SetPen 的构造函数,可以仅提供一个颜色参数,使其第二个与第三个参数取缺省值来设置画笔颜色。下面是该函数的代码:

```
void TLine::SetPen(int penSize)
{
    if(penSize<1)
        PenSize=1;
    else
        PenSize=penSize;
}

void TLine::SetPen(TColor &newColor,int penSize)
{
    // IfpenSize isn's the default(0),set PenSize to the new size.
    if (penSize)
        PenSize=penSize;

    Color=newColor;
}
```

(二)Draw 函数

由于 TLine 对象具有重绘自身的能力,故将 Paint 函数在窗口中画线的功能改由 Draw 函数来完成。看一下 Draw 函数的代码,并与第八节中的 Paint 函数比较,会发现两者几乎一致。

```
BOOL TLine::Draw(TDC &dc)const
{
    // Set pen for the dc to the values for this line
    TPen pen(Color,PenSize);
    dc.SelectObject(pen);

    // Iterates through the points in the line i.
    TPointsIterator j( * this);
    BOOL first=TRUE;

    while(j){
        TPoint p=j++;
```

```
            if(!first)
                dc.LineTo(p);
            else{
                dc.MoveTo(p);
                first=FALSE;
            }
        }
        dc.RestorePen();
        return TRUE;
    }
```

将上面代码移入 TLine 类,TMyWindow::Paint 函数变得异常简单:

```
    void TMyWindow::Paint(TDC& dc,BOOL,TRect&)
    {
        // Iterates through the array of line objects.
        TLinesIterator i(*Lines);

        while(i)
            i++.Draw(dc);
    }
```

(三)输入、输出操作符

为了处理改进的 TLine 类,输入(>>)输入(<<)操作符有必要做一定的修改。

输出操作符首先将画笔的尺寸和颜色写入文件,然后依次写入各画线点。

输入操作符首先读入画笔尺寸 PenSize 和颜色 color,并将其传给 SetPen 函数,最后依次填入所有的画线点。

5.9.2 修改 TMyWindow 类

随着 TLine 类改进,TMyWindow 类也应稍加改动:

- Pen 的数据成员包括画当前线的尺寸和颜色。
- 去掉了 SetPenSize 函数,增加了 GetPenSize 函数;该函数为用户打开一个输入对话框,供用户输入画线尺寸,然后调用 Line->SetPen 函数进行设定。
- 增加了处理 CM_PENCOLOR 事件的函数 CmPenColor。当用户选择菜单选项"Pen Color"时产生该事件。

(一)CmPenColor 函数

CmPenColor 函数应用 TClooseColorDialog 为打开一个选择画线颜色的对话框。与 TFileOpenDialog 和 TFileSaveDialog 一样,TChoooseColor Dialog 类封装了一个 Windows 通用对话框。

另外,同 TFileSaveDialog 和 TFileOpenDialog 一样,TChooseColorDialog 类的构造函数带五个参数,但是在本例中,仅需为其中的两个参数提供具体的值,其余三个参数均取其缺

省值。这两个参数一个是指向父窗口的指针，一个是对 TChoose Color：TData 对象的引用。指向父窗口的指针可用 this 指针代替，类 TChooseColorDialog 的对象由 Colors 提供。

Color 是 Colors 对象的成员，通过把某一特殊的颜色值赋给 Color，可使该颜色成为颜色选择对话框中的缺省颜色。然后设定 Color 为当前画笔的颜色，则颜色对话框便能反映出应用程序的状态。

通过将包含一系列 TColor 对象的数组赋给 colors 的成员 Custocolor，可以将这一系列不同的颜色显示在颜色对话框中的自定义颜色区。在设计自定义颜色时，用户可以使用合作自己喜爱的颜色。在本节的应用程序中，自定义颜色由具有不同灰度的单颜色值组成，其颜色由黑变白。

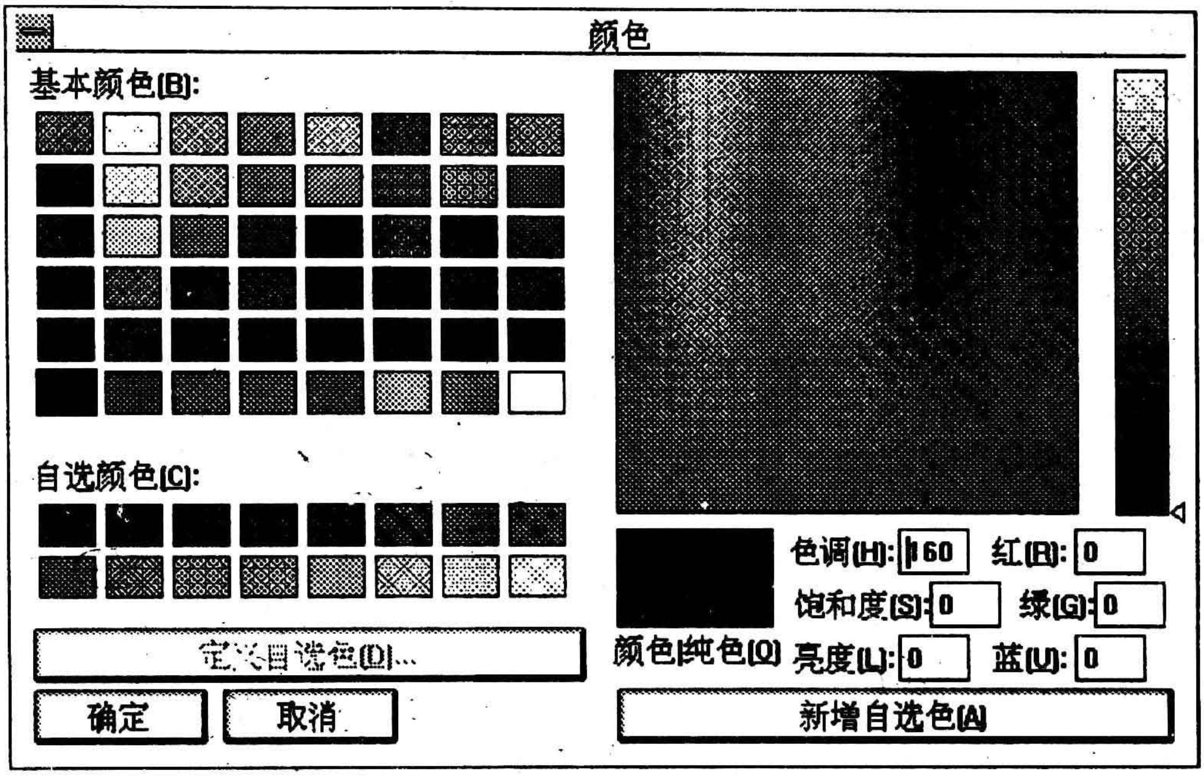
尽管颜色选择对话框有一个附加的按钮，但与文件打开对话框(TFileOpenDialog)、文件存储对话框(TFileSaveDialog)的创造与执行方法完全一样。该附加的按钮由 Windows 的通用对话框处理，因此执行该对话框的函数 Execute 仅有两个返回值：IDOK 和 IDCANCEL。如果用户选择了"Cancel"按钮，则在对话框中的所做的一切改变都将被忽略。

相反，如果用户选择了"OK"按钮，则需要将画笔颜色设定为用户新选择的颜色。即把用户选择的颜色值赋给 colors 对象的数据成员 color。color 是一个 TColor 类对象，它的类型与 SetPen 函数要求的参数类型一致。下面是 CmPenColor 函数的代码：

```
void TMyWindow:CmPenColor()
{
    TChooseColorDIalog::TData colors;
    static TColor custColors[16]=
    {
        0x010101L,  0x101010L,  0x202020L,  0X303030L,
        0x404040L,  0x505050L,  0x606060L,  0x707070L,
        0x808080L,  0x909090L,  0xA0A0A0L, 0xB0B0B0L,
        0xC0C0C0L, 0xD0D0D0L, 0xE0E0E0L, 0xF0F0F0L,
    };
    colors.Flags=CC_RGBINIT;
    colors.Color=TColor(Line->QueryColor());
    colors.CustColors=custColors;
    if(TChooseColorDialog(this,colors),Execute()==TDOK)
        Line->SetPen(colors.Color);
}
```

5.10 第十步：增加装饰

Setp10只在 InitMainWindow 函数中做了一些变动。这些变动会使应用程序看起来更吸引人，更直观。在这一节，将增加生个由位图按钮配件组成的控制条和一个显示当前菜单选择的状态条。程序运行结果如下图：



这一步有四个改进：

- 用 TDecoratedFrame 代替 TFrameWindow 作主窗口；
- 创建状态条并插入装饰框架窗口；
- 创建控制条及其按钮配件，并插入装饰框架窗口；
- 增加资源，如字符串表（用来描述每个菜单选项的功能）和控制配件位图。

5.10.1 修改主窗口

将 TFrameWindow 改为 TDecoratedFrame。由于 TDecoratedFrameWindow 基于 TFrameWindow，所以只要 TFrameWindow 可以使用的地方，TDecoratedFrame 也能使用。此时，只要创建 TDecoratedFrame 对象并作为参数传给 SetMainWindow 函数。

TFrameWindow 和 TDecoratedFrame 的构造函数几乎是一样的，差别只在第四个参数。

TFrameWindows 的第四个参数是个 BOOL 参数，该参数通知框架窗口是否收缩成和客户窗口一样的大小。

TDecoratedFrame 的第四个参数也是 BOOL 参数。该参数表示装饰框架窗口是否要跟踪菜单选项。菜单跟踪在消息条或状态条里显示有关当前选中的菜单项或按钮的文字描述。该参数如果为 TRUE，则必须提供消息条或状态条，否则应用程序第一次运行时便会崩溃，并试图给消息条或状态条发送一条消息。

如果要用状态条，必须在资源文件中包含它的资源。这些资源包含在 INCLUDE、OWL 目录下的 STATUSBA.RC 文件中。

其他区别是：在装饰框架成为主窗口之前，要做一些准备工作。诸如增加状态条和控制条，然后将它设为主窗口。

5.10.2 创建状态条

用 TStatusBar 类创建状态条。TStatusBar 是基于 TMessageBar 类的，而 TMessageBar 又是基于 TGadgetWindow 的。消息条和状态条都显示文本信息。但状态条的选项要多。例如，在一个状态条里有多个文本配件，模式提示栏(如 Insert 模式和 OverWrite 模式)。

TStatusBar 的构造函数有五个参数，常用前面两个参数，余下三个参数用缺省值。

■ 第一个参数是指向状态条的父类窗口的指针。此时，用 frame，它是指向刚创建的装饰框架窗口的指针。

■ 第二个参数是 TGadget::TBorderstyle 枚举。它是 None，Plain，Raised，Recessed 或 Embossed 之一。这个参数决定状态条的风格。缺省取 Recessed。

■ 第三个参数是 TModeIndicator 枚举量。它决定状态条显示何种模式的键盘。可以是 ExtendSelection，CapsLock，ScrollLock，Overtype 和 RecordingMacro 之一或几个组合。这个参数缺省为0，即无键盘模式。

■ 第四个参数是 TFont *。这个参数包含显示在状态条里的字体。缺省是 TGadgetWindowFont。

■ 第五个参数是 TModule *，缺省取0。

这儿是状态条的构造函数。

```
TStatusBar * sb=new TStatusBar(frame,TGadget::Recessed);
```

一旦创建了状态条，就要将它插入框架窗口。

5.10.3 创建控制条

一、构造 TControlBar

TControlBar 的构造函数带四个参数，一般只用前两个，余下参数取缺省值：

■ 第一个参数是指向父窗口的指针，跟状态条一样，用 frame 将装饰框架窗口作为控制条的父窗口。

■ 第二个参数是 TTileDirection 枚举。该枚举量有两个值：Horizonal 和 Vertical。这个参数提示状态条如何安放控制。缺省用 Horizotal，水平安放控制。

■ 第三个参数是 TFont *，指示状态条使用的字体。缺省值是 TGadgetWindowFont。

■ 第四个参数是 TModule *，缺省为0。

控制条的构造函数如下：

```
TControlBar * cb=new TControlBar(frame);
```

二、创建按钮配件

按钮配件用作控制条按钮，它们将位图按钮和事件标识符联系起来。按下按钮配件，它便送出一个事件标识符。可以设置成按下一个配件就好像执行了一次菜单选择。这一部分将

讲述如何设置按钮配件来替换当前菜单选项。

通过构造 TButtonGadget 来创建按钮配件。构造函数共六个参数，一般只用前三个。

- 第一个参数是一个 TResId 的引用对象。这个参数是按钮配件的位图资源标识符。一般对位图大小没有限制，但在实际应用时，还应作点考虑。控制条的高度是基于包含在控制条内控制对象的尺寸，如果按钮配件很大的话，则控制条也会很大。
- 第二个参数是按钮配件的标识符。一般情况下，配件标识符，事件标识符和位图标识符是同一个值。例如，替代 File 菜单中 New 命令的按钮的位图资源标识符为 CM_FILEOPEN，配件标识符也是 CM_FILEOPEN，选择按钮时传递事件 CM_FILEOPEN，选择按钮时传递事件 CM_FILEOPEN。

  在资源文件中，位图和事件使用同一个标识符，这在编写代码时较为容易。但这并不是一种规定，你可以随心所欲地给位图和事件命名不同的标识符。唯一规定是：必须定义事件标识符和处理程序，资源标识符要合法。

  还要注意，在应用程序的字符串资源表中有很多项，它们和按钮配件及事件具有相同的 ID 值。如果一个字符串和按钮配件有相同的标识符，则按下按钮时，该字符串就会显示在状态条内。
- 第三个参数是 TType 枚举量，标识何种类型的按钮。有三个可选值：Command，Exclusive 和 NonExclusive。在这个应用中，所有按钮均是命令按钮。缺省值是 Command。
- 第四个参数是 BOOL 量，指明按钮是否激活。缺省值是 FALSE。
- 第五个参数是 TState 枚举量，指明按钮的初始状态。可以是 Up，Down 和 Indeterminate。缺省值是 Up。
- 第六个参数是 BOOL 值，指明按钮的重复状态。如果重复状态为 TRUE，则当按钮按下和保持时，按钮重复。缺省为 FALSE。

三、分隔配件

还有一种配件称为分隔配件。通常配件在创建时后一个总是紧跟着前一个放在右边。分隔配件在两个配件之间留出一点空间。这样便于将配件分组放置，放在预定的地方。

分隔配件包含在 TSeparatorGadget 类中，这个类较简单，只带一个 int 参数，缺省时，这个值为6，表示分隔配件占6个像素的空间。

四、将配件插入控制条中

一旦创建了配件，就需要将它插入控制条中。由于控制条从类 TGadgetWindow 派生而来，所以可以有配件。TGadgetWindow 提供基本功能来使用窗口里的配件。TControlBar 加强了这种功能。

用 Insert 函数将配件插进控制条。这个函数是 TControlBar 从 TGadgetWindow 中继承而来的（稍后，使用 TDecoratedFrame 中的 Insert 函数）。这个函数带三个参数：

- 第一个参数是 TGadget 或 TGadget 派生类的引用；
- 第二个参数是 TPlacement 枚举量，其值是 Before 或 After。这个参数提示配件是放

在其他兄妹配件之前还是之后。缺省值为 After。如果没有定义其他兄妹配件，该值无影响。

■ 第三个参数定义兄妹配件。这些兄妹配件必须已经插进控制条，缺省为0。

在手册中的应用程序例子里，创建配件和插入控制条是在一步里完成的。这是插入控制条的代码：

```
cb-> Insert( * new TButtonGadget(CM_FILENEW,CM_FILENEW,
            TButtonGadget::Command));
cb-> Insert( * new TButtonGadget(CM_FILEOPEN,CM_FILEOPEN,
            TButtonGadget::Command));
cb-> Insert( * new TButtonGadget(CM_FILESAVE,CM_FILESAVE,
            TButtonGadget::Command));
cb-> Insert( * new TButtonGadget(CM_FILESAVEAS,CM_FILESAVEAS,
            TButtonGadget::Command));
cb-> Insert( * new TSeparatorGadget);
cb-> Insert( * new TButtonGadget(CM_PENSIZE,CM_PENSIZE,
            TButtonGadget::Command));
cb-> Insert( * new TSeparator Gadgets);
cb-> Insert( * new TButtonGadget(CM_ABOUT,CM_ABOUT,
            TButtonGadget::Command));
```

注意：这些配件只是替换已经有的菜单命令，它提供了一种快速使用那些需要频繁选择菜单命令的方法。当然，不一定非要用控制条中的配件来作为菜单命令的替代品或快捷键。通过 *TType* 参数，可以在控制条里设置互锁按钮(在配件组中用 *Exclusive* 参数)，复选框(用 *NonExclusive* 参数)。

5.10.4 在装饰框架里插入对象

为 TDecoratedFrame 窗口构造装饰件后，接下来就是要把这些装饰件插入装饰框架窗口，并使该窗口成为应用程序的主窗口。

将装饰件放进装饰框架和将配件放入控制条原理是一样的，TDecoratedFrame::Insert 函数带两个参数：

■ 第一个参数是 TWindow 或 TWindow 派生类对象引用。这个窗口对象就是装饰件。本例中，窗口派生类对象就是 TStatusBar 对象和 TControlBar 对象。

■ 第二个参数是 TLocation 枚举量，这个值是 Top，Bottom，Left 或 Right 之一，表示配件放在装饰框架窗口中的位置。

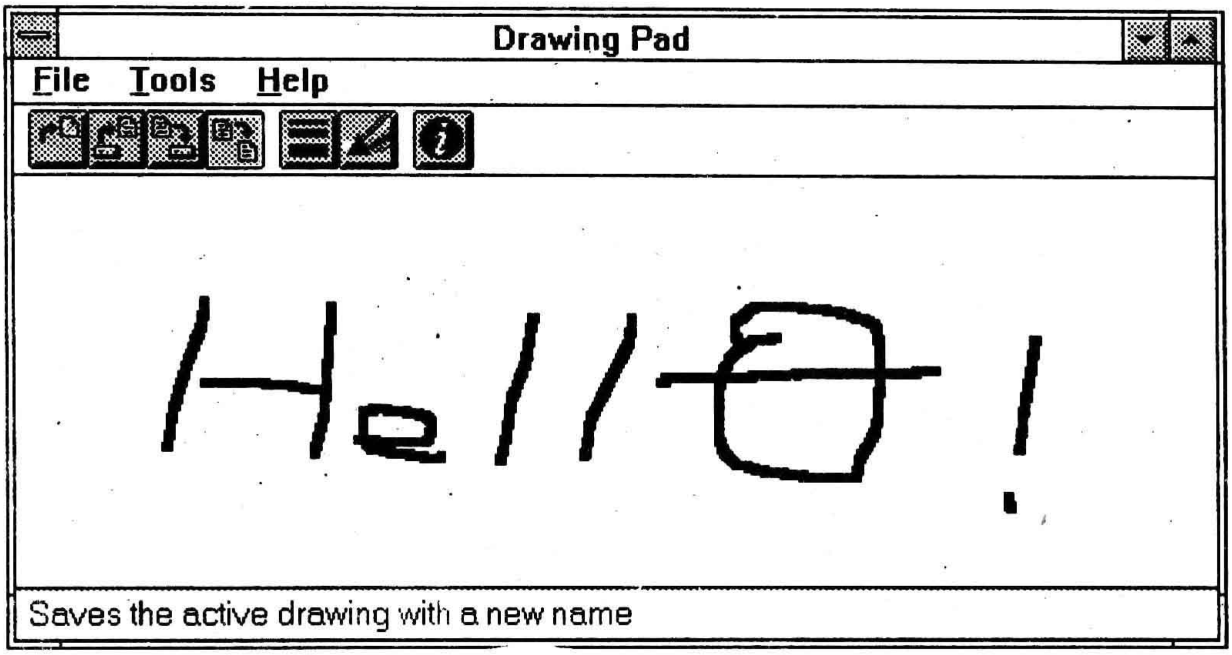
下列代码用于将装饰件放入装饰框架：

```
// Insert the status ar and control bar into the frame
    frame-> Insert ( * sb,TDecoratedFrame::Bottom);
```

```
frame->Insert(*cb,TDecoratedFrame::Top);
```

接下来就是将 frame 设为主窗口和菜单：

```
// Set the main window and its menu
    SetMainwindow(frame);
    GetMainWindow()->AssignMenu("COMMANDS");
```



5.11 第十一步:转向 Doc/View 模块

step11介绍 Doc/View 编程模块,该模块是基于将数据和数据接口分开来的原理设计的。从实质来看,数据封装在文档对象中,用户利用浏览器对象显示数据并对数据进行操作。文档对象从 TDocument 在类派生,浏览器对象从 TView 类派生。

Doc/View 模块显示和处理数据的方式比将数据和数据接口封装在一个类中这种处理方式更灵活。应用 Doc/View 模块,定义文档类来包含任何类型的数据,诸如纯文本文件,数据库文件,或示例中的画图文件。然后创建很多的浏览器,每个浏览器用不同的方式显示同一种数据,或者让用户用不同的方式和数据交互。

step11只是简单的将应用程序从当前模块转代成 Doc/View 模块。step11代码和 step10代码差别很大,但运行结果和 step10的运行结果看起来几乎一样。

5.11.1 应用程序原码的结构

setp11的原码分成四个原文件：

- SETP11.CPP 包含应用程序对象和成员定义及 OWLMain 主函数
- SETP11.RC 包含应用程序事件标识符,框架窗口和装饰件的资源,对话框和菜单。
- SETP11DV.CPP 包含 TLine 类,文档类 TDrawDocument,浏览器 TDrawView,以及每个类相关的成员函数定义。

■ STEP11DV.RC 包含浏览器类对象处理的事件标识符和浏览器资源。

通过这种分类，将 Doc/View 代码分成文档代码及其相关的浏览器代码，而应用程序代码则支持文档类和浏览器类整个框架，而不用涉及到 Doc/View 模块的内部功能。这对代码重用大有好处。

5.11.2 Doc/View 模块

Doc/View 模块基于 ObjectWindows 的三个类：

■ TDocument 类封装和控制数据存取。文档对象处理从相关的浏览器对象输进的数据。同一时间，一个文档对象可以和多个浏览器对象关联(为简单起见，本例中文档对象只和一个浏览器对象关连)。

■ TView 类提供文档对象和用户之间的接口。浏览器对象控制来自文档对象的数据如何在屏幕上显示。一个浏览器对象一次只能和一个文档对象关联。

■ TDocManager 类协调文档对象和浏览器对象之间的关系。文档管理器提供缺省的文件(File)菜单及每个菜单选项的缺省处理。它还维护一个文档模板链表，这个文档模板规定了文档类和浏览器类之间的关系。TDocument 和 TView 类将文档对象和浏览器对象的功能高度抽象化，因而必须在自己的文档类和浏览器中将这些功能具体化。另一个必须做的是创建文档管理器并和应用程序对象挂钩，同时为文档管理器提供文档模板。下一部分讨论这几个步骤。

5.11.3 TDrawDocument 类

TDrawDocument 类从 ObjectWindows 类 TFileDocument 派生出来，而 TFileDocument 又从 TDocument 派生而来。TDocument 类提供了一些输入和输出函数，这些虚拟函数返回"哑元"值，不做实质性的事。TFileDocument 提供了一些基本函数以流的形式存取数据文件。

TDrawDocument 用 TFileDocument 中的函数来访问存储在文件中的直线数据。它用 TLines 数组来存这些数据。通过叫 Lines 的指针来引用这个数组。

一、创建和撤销 TDrawDocument

TDrawDocument 的构造函数带一个指向 TDocument * 父类文档的指针。这个父文档可以是其他的文档，包含在该文档中的数据被当作父文档的一部分处理。构造函数把父类文档指针传给 TFileDocument。构造函数也将数据成员 Lines 初始代为0。

析构函数删除 Lines 指向的 TLines 对象。

二、存储直线数据

文档类负责控制图形中的数据存取，必须想办法存这些数据。前一步创建的 TLine 类和 Tlines 数组代码在本例中可以拿来用。每个文档的直线数据存在 TLines 数组中，文档通过保持的数据成员 TLines * 指针 Lines 来存取这些数据。

TPoints 和 TLines 类数组及其包容类，TLines 类都在 STEP11DV.CPP 中定义。这些类

是 Dco/view 模块中文档类的组成部分。这些代码从 Step10照搬过来。

三、完成 TDocument 的虚拟函数

TDrawDoçument 需要完成几个从 TDocument 继承来的虚函数。这几个函数提供流存取功能，执行对文档的修改，或者放弃自文档上次存盘以来所作的修改。

四、打开和关闭图形

尽管 TFileDocument 提供了流输入/流输出的基本功能，但它并不知道怎样从直线中读了数据。重载 Open 和 Close 函数来提供这种功能。

Open 函数如下：

```
BOOL Open(int mode,const char far * path=0);
```

这里：

- mode 参数是文件打开模式。本例中忽略这个参数；每次文件打开模式相同（根据 of Read 标志）
- path 包含文档路径，如果指定了路径，文档的当前路径就是这个指定的路径，如果没有指定路径(即 path 用缺省值，就用当前路径。在创建文档流时要用到路径。

Open 函数和前面例子中用的 OpenFile 函数相同，区别在于：

- Open 函数为文档对象创建 Tlines 类数组，在前面的例子中，这一步是在 TMyWindow 构造函数中进行的，因为 TMyWindows 负责包含所有的 TLine 对象。现在由文档类负责所有的 TLine 对象，所以在数据读进之前必须分配空间。
- 如果 path 参数指定，Open 函数用 SetDocPath 函数将文档对象路径设置为 path。
- Open 函数检测文档是否有路径。如果文档没有路径，那么就是一个新文档，此时就不必从文件中读进数据；如果文档有路径，Open 调用 InStream 函数，该函数定义在 TFileDocuemnt 中，返回参数 TInStream * 。

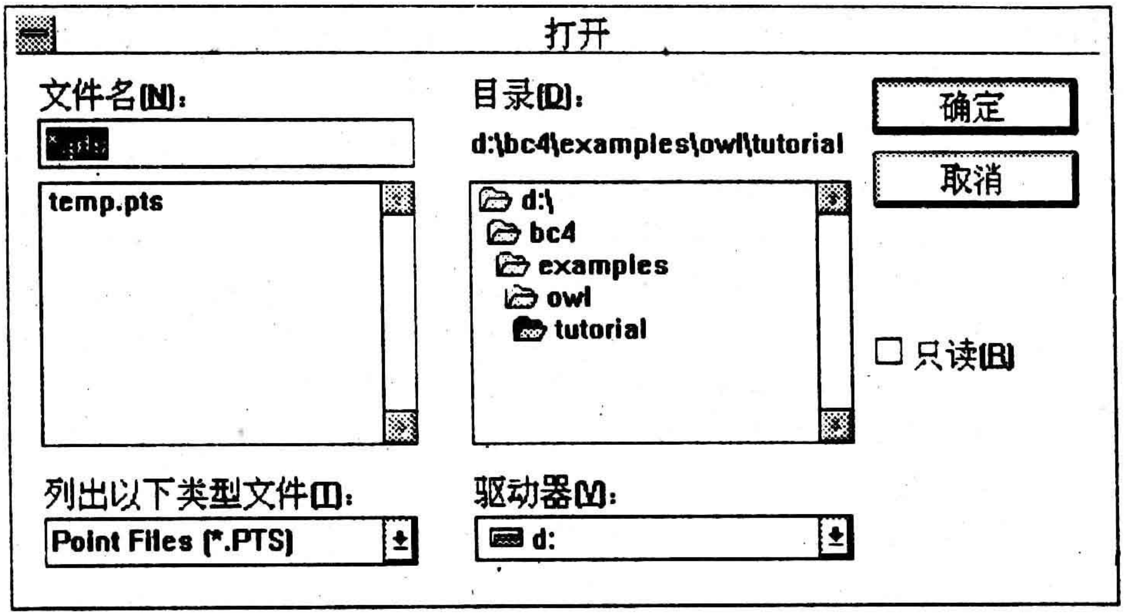
  TInStream 是 Doc/View 类使用的标准输入流类，字从 TStream 和 istream 派生。TStream 是个抽象基类，文档类利用它来标准流。TInStream 本质上是标准 istream，是专为 Doc/View 模块改做的。还有对应的 TOutStream 类（从 TStream 和 Ostream 派生）。在创建 Commit 函数时将看到使用 TOutStream 类。
- 输入流创建之后，就读取数据并放在由 Lines 指向的 TLines 类数组中。所有数据读进之后，输入流就被删除了。
- Open 函数然后调用 SetDirty 函数，传送 FALSE 参数。SetDirty 和另一个函数 is Dirty 具有相同的功能，和前几步例子中 IsDirty 标志功效相同。如果文档中数据修改之后还没有存盘或者这些修改还没有生效，我们就说该文档是“脏”的。
- Open 函数最后要返回值，如果文档打开成功，则返回 TRUE。

Open 函数的代码看起来如下：

```
BOOL TDrawDocument::Open(int/ * mode * /,const char far * path)
{
```

```
    Lines=new TLines(5,0,5);
    if(path)
        SetDocPath(path);
    if(GetDocPath()){
        TInStream * is =InStream(ofRead);
        if(!is)
            return FALSE;
        unsigned numLines;
        char fileinfo[100];
        *is >> numLines;
        is-> getline(fileinfo,sizeof(fileinfo));
        while(numLines--) {
            TLine line;
            *is >> line;
            Lines-> Add(line);
        }
        delete is;
    }
    SetDirty(FALSE);
    return TRUE;
}
```

弹出的对话框如图：



关闭画图操作相对容易一些。函数 close 放弃文档中的数据并对文档进行清理。本例中，函数删除 Lines 指向的 TLines 数组并返回 TRUE 以下是 Close 函数的代码。

```
BOOL TDrawDocument::Close()
{ delete Lines;
  Lines=0;
```

```
    return TRUE;
}
```

将 Lines 设置为0;在构造函数和文档关闭之后都要这么做,这样很容易知道文档是否打开。如果文档已经打开,Lines 就会指向 TLines 数组,其值就不会为0,所以将 Lines 设置为0时很容易检测文档是否打开。函数 IsOpen 可以在文档对象之外检测文档是否打开:

```
BOOL IsOpen() {retun Lines!=0;}
```

五、存储和放弃文档的修改

TDocument 提供两个函数处理文档修改后的存储和放弃工作。

- Commit 函数执行对文档的修改,然后将数据存在永久介质上。通过和文档相关联的流利器来修改文档中的数据。Commit 函数带一个 BOOL 参数,如果参数为 FALSE,则仅当文档是"脏"的时,Commit 函数才存数据;如果参数为 TRUE,则 Commit 函数将数据全写一次;参数缺省为 FALSE。
- Revert 函数放弃文档浏览器中的任何修改,然后迫使浏览器重新调入文档中的数据并显示;Revert 函数也带一个 BOOL 参数。如果该参数为 TRUE,则浏览器清除自己的窗口,且不从文档中重调数据。缺省时参数为 FALSE。

对 TDrawDocument,每次在浏览器窗口中画一条线时,文档都会更新。在 TDrawDocument 类中,Commit 只是将数据存入文件中。

Commit 检测文档是否是"脏"的,如果文档不"脏",且参数(BOOL force)为 FALSE,则 Commit 返回 TRUE,表示这次操作是成功的。

如果文档是"脏"的,或者 force 参数为 TRUE,则 Commix 存数据。存数据的过程和前一步 SaveFile 函数是一样的。

Commit 函数调用 OutStream 函数打开一个输出流。该函数定义在 TFileDocument 中,且返回一个 TOutStream * 。Commit 然后将数据写入输出流。这一过程几乎和较老的函数 SaveFile 相当。

将数据写入输出流后,Commit 将 IsDirty 标志设为 off,这是通过调用 SetDirty (FALSE)完成的。然后 Commit 返回 TRUE,表明操作成功。

Commit 代码如下:

```
BOOL TDrawDocuemnt::Commit(BOOL force)
{
    if(!IsDirty() && ! force)
        return TRUE;
    TOutStream * os=OutStream(ofWrite);
    if( ! OS)
        retrn FALSE;
    // Write the number of lines in the figure
    *os << Lines-> GetItemsIncontainer();
```

```
    // Append a description using a resource string
    *os <<' ' << string(*GetDocManager().GetApplication(),IDS_FILEINFO)<<'\n';
    // Get an iterator for the array of lines
    TLinesIterator i(*Lines);
    // While the iterator is valid (i.e. we haven't run out of lines)
    while(i) {
        // Copy the current line from the iterator and increment the array.
        *os << i++;
    }
    delete os;
    SetDirty(FALSE);
    return TRUE;
}
```

该代码中,有一行以前还没遇到:

```
// *os<<' '<< string(*GetDocManager().GetApplication(),IDS_FILEINFO)<<'In';
```

它是用了 ANSI 司类 String 的特殊构造函数:

```
String(HINSTANCE instance,UINT id,int len=255);
```

这个构造函数从 Windows 应用程序中获取一个字符串资源。传递一个 HINSTANCE 作为 String 构造函数的第一个参数,来指明应用程序。本例中通过文档管理员来获取当前应用程序实例;函数 GetDocManager 返回指向文档的文档管理员指针,同样,GetApplication 返回包含文档管理器的一个应用程序指针。通过 TModule 类中的转换操作符转换成 HINSTANCE。String 的第二个参数是定义在 STEP11DV.RC 中字符串资源的资源标识符。这个字符串含有能标识创建文档的应用程序的版本信息。

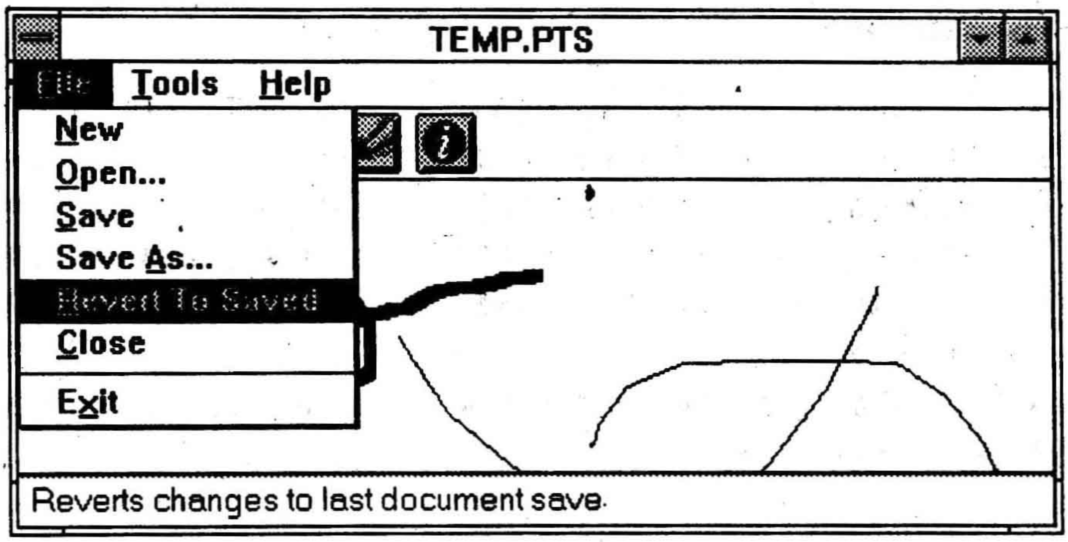
Revert 函数带有一个 BOOL 参数,表明文档浏览器是否需要用文档中的数据来刷新显示。Revert 调用 TFileDocument 的 Revert 函数,它反过来又调用 TDocument 的 Revert 函数。基类函数调用 NotifyViews 函数响应 vnRevert 事件。NotifyView 函数的第二个参数设置传给 TDrawDocument::Revert 函数的参数 TFileDocument::Revert 设 IsDirty 为 FAISE,然后返回。如 TFileDocument::Revert 返回 FALSE,TDrawDocument 也应该返回 FALSE。

如果 TFileDocument::Revert 返回 TRUE,TDrawDocument 函数就必须检查传给 Revert 的参数。如果参数为 FALSE,(意味着浏览器需要刷新),Revert 就调用 Open 函数打开文档文件,重新调入文档数据,并显示。

Revert 函数的代码如下:

```
BOOL TDrawDocuemnt::Revert(BOOL clear)
{
    if(!TFileDocument::Revert(clear))
        return FALSE;
    if(!clear)
        Open(0);
```

```
    return TRUE;
}
```



六、存取文档数据

有两种方式来存取文档中的数据：增加一条线(例如用户在浏览器中画一条新线)及获取文档中一条线的引用(如窗口重绘时获取每条线的引用)。增加两个函数 AddLine 及 GetLine 来完成这些任务。

AddLine 向文档的 TLines 数组增加一条线，这条线是作为 TLine& 传给 AddLine 的，将线加入数组之后，AddLine 将 IsDirty 设为 TRUE，然后返回这条刚加入线的索引号。

AddLine 函数的代码如下：

```
int IDraw Document::AddLine(TLine& line)
{
    int index=Lines->GetItemsInContainer();
    Lines->Add(line);
    SetDirty(TRUE);
    return index;
}
```

GetLine 函数带一个 int 参数。这个参数是所需线的索引。GetLine 首先要检查文档是否打开，如果没有打开，就试图打开文档。如果文档没有打开，而 GetLine 又不能打开，则它返回0，表示没有找到一个合法的文档来从中获取线。

一旦找到合法的文档，必须检测该文档以保证索引号不是太高。将索引号和 GetItemInContainer 函数的返回值比较，只要索引较小，就能返回一个指向 TLine 对象的指针。下面是 GetLine 的代码：

```
const TLine * TDrawDocument::GetLine(unsigned int index)
{
    if (!IsOpen() && ! Open(ofRead|ofWrite))
        return 0;
```

```
    return index <Lines->GetItemsInContainer()?&(*Lines)[index]:0;
}
```

5.11.4 TDrawView 类

TDrawView 类从 ObjectWindows 的 TWindowView 类派生，而 TWindowView 又从 TView 和 TWindow 派生而来，TView 没有继承窗口的能力，一个 TView 派生类要么增加一个窗口成员或指针，要么在窗口类中和浏览器类混合，方能得到这些继承能力。

TWindowView 采用后一种方法；将 TWindow 和 TView 类结合起来，组成一个新类，该类提供了基本的窗口和浏览功能。通过从这个通用类派生出新类 TDrawView 类，它只需要增加处理 TDrawDocument 类的功能。

TDrawView 类和前几步中 TMyWindow 相同，实际上，TMyWindow 中很多函数几乎没有经过修改就用到 TDrawView 类中。

一、TDrawView 的数据成员

TDrawView 中保护的数据成员如下：

```
TDC *DragDC;
Tpen *Pen;
TLine *Line;
TDragDocument *DrawDOC;
```

有三个数据成员我们已经较熟悉。DragDC，Pen 和 Line 在 TDrawView 和 TMyWindow 中的功能相同。

尽管没有浏览器与之关联，文档也能存在，但没有文档，浏览器却不能存在。浏览器必须和一个存在的文档关联。在 TDrawView 构造时，它就和它的文档附着在一起，并通过一个指向 TDrawDocument 的 DrawDoc 指针来跟踪它的文档。基类 TView 有一个叫 Doc 的指向 TDocument* 的成员。实际上，在基类构造过程中，Doc 指向 TDrawDocument 对象，该对象传给 TDrawView 构造函数。增加 DrawDoc 指针来保证存取文档指针时强制类型匹配。

二、创建 TDrawView 类

TDrawView 构造函数带两个参数；一个是 TDrawDocument& 引用(引用浏览器相关的文档)，另一个是 TWindow * 指针(指向父窗口的指针)。如果不提供值，父窗口缺省为0，构造函数将这两个参数传给 TWindowView 的构造函数。并把 DrawDoc 成员初化成指向文档的指针。该文档是作为第一个参数传递过来的。

构造函数还将 DragDc 设为0，并用 TLine 对象初始化 Line。

最后构造函数设置浏览器菜单。可以用 TMenuDescr 类来从菜单资源中建立菜单描述符。下面是 TMenuDescr 构造函数：

```
TMenuDescr(TResId id,int fg,int eg,int cg,int og,int wg,int hg);
```

这里：

- id 是菜单资源的资源标识符；

■ fg 是文件(File)菜单的菜单组号；

■ eg 是编辑菜单(edit)的菜单组号；

■ cg 是容器菜单(Container)的菜单组号；

■ og 是对象菜单(object menu)的菜单组号；

■ wg 是窗口菜单(Window)的菜单组号；

■ hg 是帮助菜单(Help)的菜单组号；

尽管每个组有特别的名字，但这些名字仅代表菜单组的通用名字，每个组代表的菜单不一定要有这个名字，文档管理器提供缺省的 File 菜单，但其他菜单可在菜单资源中设置。

如果其中一个菜单组参数为0，则表示菜单资源没有给这个组提供菜单。整个菜单组参数表示的菜单组总项数应该等于或少于菜单资源中能够获得的菜单组的项数。

本例中，浏览器提供一个 IDM_DRAWVIEW 菜单资源。

该资源定义在 STEP11DV. RC 中。该菜单称 Tools 菜单，它有两个选项：Pen Size 和 PenClor。如果要把 Tools 菜单插在主菜单的第二个菜单位置上，eg 参数(第二个菜单组参数)应该为1。其余的参数都为0。

调用 TView 函数 SetViewMenu 函数来安装菜单描述符，将它作为浏览器的菜单。SetViewMenu 带一个 TMenuDescr * 参数。将它设置为浏览器的菜单。当创建浏览器时，这个菜单组合在应用程序菜单中。

设置浏览器菜单的调用如下：

```
SetViewMenu(newTMenuDescr(IDM_DRAWVIEW,0,1,0,0,0,0));
```

浏览器的析构函数删除 DragDC 引用设备上下文，以及 Line 引用的 TLine 对象。

三、返回类名

每一个浏览器都应该定义函数 StaticName，该函数不带参数，返回 Static const char far * 。该函数应该返回浏览器的类名。下列是 StaticName 函数的代码：

```
Static const Char far *  StaticName() { return "Draw View"; }
```

四、保护的函数

TDrawView 提供了几个保护性的成员函数。

GetPenSize 函数和 TMyWindow 的函数 GetPenSize 相同。

该函数打开一个 TInputDialog 对话框来获得用户输入的笔的大小；然后修改窗口的画笔尺寸并调用 SetPen 函数。

Paint 函数和 TMyWindow 的 Paint 函数有一点区别，但基本功能差不多，抛弃用"iterator"遍历数组中直线的做法，TDrawView::Paint 调用 GetLine 函数。从 GetLine 返回的值赋给 const TLine * 指针 Line 如果 Line 不等于0(即 GetLine 返回一个合法值)，Paint 便调用直线的 Draw 函数。记住从第10步以来 TLine 类没有变化。

Paint 函数的功能代码如下：

```
void TDrawView::Paint(TDC  * dc,BOOL,TRect&)
```

```
// 遍历直线对象数据
int i=0;
const TLine * line;
while ((line=DrawDoc->GetLine(i++)) ! =0)
    Line ->Draw(dc);
}
```

五、TDrawView 中的事件处理

TDrawView 处理许多由 TMyWindow 类处理的事件。剩余的 TDrawView 没有处理的事件由应用程序对象和文档管理器处理。稍后将讨论。

另外,TDrawView 处理两条消息,VN_COMMIT 和 VN_REVERT 这些浏览器通知消息由浏览器文档传来(当文档的 Commit 和 Reuert 函数调用时)。

下面是 TDrawView 的响应表:

```
DEFINE_RESPUNSE_TABLE1(TDrawView,TWindowView)
    EV_WM_LBUTTONDOWN,
    EV_WM_RBUTTONDOWN,
    EV_WM_MOUSEMOVE,
    EV_WM_LBUTTONUP,
    EV_COMMAND(CM_PENSIZE,CmPenSize),
    EV_COMMAND(CM_PENCOLOR,CmPenColor),
    EV_VN_COMMIT,
    EV_VN_REVERT,
END_RESPONSE_TABLE;
```

下表中的函数和 TMyWindow 类中相应的函数几乎一样。若有改动,在表的右列注明:

函　数	TDrawView 版
EvLButtonDown	不设置 IsDirty,这在 EvLButtonUp 中设置
EvRButtonDown	不变
EvMouseMove	不变
EvLButtonUp	检测鼠标左键按下后,是否移动;如果移动,调用文档的 AddLine 函数来加入点
CmPenSize	不变
CmPenColor	不变

VnCommit 函数永远返回 TRUE,在复杂的应用程序中,该函数应该把缓存(cached)数据加进文档中,但在本例中,画每一条线时,数据都要加到文档中。

VnRevert 函数使显示区域中的数据无效,清除数据并重复窗口中的图形,然后返回 TRUE。

5.11.5 定义文档模板

一旦定义了文档类及与之相伴的浏览器类,就必须将两者关联起来,以便协同工作。文

档类和浏览器类之间的关联就是文档模板类。文档管理器使用文档模板来决定打开什么样的浏览类来显示文档。

用 DEFINE_DOC_TEMPLATE_CLASS 宏定义文档模板类。这个宏带三个参数，第一个参数是文档类名，第二个参数是浏览器类名，第三个参数是文档模板类名，创建 TDrawDocument 和 TDrawView 类的模板代码如下：

```
DEFINE_DOC_TEMPLATE_CLASS(TDrawDocument,TDrawView,
                                    DrawTemplate);
```

一旦创建了文档模板类，就可以创建该类的实例。可以给实例取一个比较有意义的名字。文档模板的构造函数如下：

```
TplName name (const char far * desc,const char far * filt,
              const char far * dir ,const chaf far * ext,longflags=0);
```

这里：

- TplName 是定义的文档模板类名；
- name 是创建的实例名
- desc 是模板的文本描述；在文件打开和存盘对话框中显示为文件种类；
- filt 是一个字符串，用来过滤当前目录中的文件名；它可以是合法的 DOS 正则表达式。
- dir 是检查文档文件时的缺省目录；
- ext 是缺省文件扩展符，如果没有定义扩展符，就用该扩展符，为0时表示没有缺省后缀。
- flags 是文档打开或创建的模式：是下列值之一或几个：

dtAutoDelete，dtNoAutoView，dtSingleView，dtAutoOpen，dtConfirm，dtHidden，第九章描述了这个标识符。

TDrawDocument 及 TDrawView 的模板实例如下：

```
DrawTemplate DrawTpl("Point Files ( *.pts)",
                     "*.PTS",0,"PTS",dtAutoDelete|dtUpdateDir);
```

5.11.6 实用程序中支持 Doc/View

STEP11.CPP 包含应用程序和定义主窗口的代码，应用程序对象为 STEP11DV.CPP 中定义的 Doc/View 类提供了一个框架。本节讨论对 TMyApp 类的修改，以使其支持新 Doc/View 类。OwlMain 保持不变。

一、InitMainWindow 函数

InitMainWindow 函数要做少许变动以支持 Doc/View。

- TDecoratedFrame 构造函数将 TMyWindow 构造函数的客户窗口设为0。客户窗口在 EvMenuDescr 调用。SetMenuDescr 从 TFrameWindow 继承而来，带一个 TMenuDe-

scr 参数。TMenuDescr 对象要用 COMMANDS 菜单资源创建，调用如下：

```
GetMainWindow()->SetMenuDescr(TMenuDescr
                                        ("COMMANDS",1,0,0,0,0,1));
```

- 增加 SetDocManager 调用，该函数设置 TApplication 类的 DocManager 成员，该函数带一个 TDocManager * 参数。
- TDocManager 构造函数带一个参数，该参数是、个或多个用或(OR)操作组成的值。唯一需要的标志是 dmSDI 或 dmMDI。这些标志让文档管理器监视单文档接口应用程序(dmSDI)或多文件接口应用程序(dmMDI)。

本例中，创建了一个 SDI 应用程序，所以指定 dmSDI 标志。另外还要指定 dmMenu 标志，该标志指定文档管理器提供缺省菜单。

SetDocManager 函数调用如下：

```
SetDocManager(newTDoCmanager(dmSDI|dmMenu));l
```

二、InitInstance 函数

创建主窗口之后，就要调用 InitInstance 函数，由于要在 InitInotance 函数中调用其他函数，所以要重写 InitInstance 函数。该函数应该先调用 TApplication::InitInstance，该函数接着调用 InitMainWindow，来构造主窗口对象，并生成主窗口。

调用基类 InitInstunce 之后，接着调用主窗口函数 DragAcceptFiles 函数，该函数带 TRUE 参数，保证主窗口接受拖进其中的文件。在下一部分，讲述应用程序的响应表如何处理拖放功能。

在应用程序起动之后，用户就要在窗口中画图，这时要调用文档管理器的 CmFileNew 函数，该函数在主窗口中创建一个没有标题的文档和浏览器。

InitInstance 函数代码如下：

```
void
TMyApp::InitInstance()
{
    TApplication::InitInstance();
    GetMainWindow()->DragAcceptFiles(TRUE);
    GetDocManager()->CmFileNew();
}
```

三、给 TMyApp 增加函数

TMyApp 中增加了几个函数。重载 TApplication 的 InitInstance 函数，增加响应表，将 TMyWindows 中的 CmAbout 函数放在这里；在响应表中增加 EV—WM_DROPFILES 宏，并增加处理该事件的函数 EvDropFiles；它是处理 WM_OVLVIEW 事件，表示浏览器要求该消息，有两个函数来处理该消息：EvNewView 处理 WM_OWLVIEW 消息(带 dnCreate 参数)，EvCloseView 也处理该消息(带 dnClose 参数)。

TMyApp 类声明如下：

```
class TmyApp:public TApplication{
    public:
        TMyApp():TApplication(){}

    protected:
        // Override methods of TApplication
        void InitInstance();
        void InitMainWindow();
        // Event handlers

        void EvNewView(TView& view);
        void EvCloseView(TView& view);
        void EvDropFiles(TDropInfo dropInfo);
        void CmAbout();
    DECLARE_RESPONSE_TABLE(TMyApp);
};

DEFINE_RESPONSE_TABLE1(TmyApp,TApplication)
    EV_OWLVIEW(dnCreate,EvNewView),
    EV_OWLVIEW(dnClose,EvCloseView),
    EV_WM_DROPFILES,
    EV_COMMAND(CM_ABOUT,CmAbout),
END_RESPONSE_TABLE;
```

四、CmAbout 函数

CmAbout 函数和 TMywindow 版该函数几乎一样，唯一区别是 CmAbout 函数不再包含父类窗口。它直接调用 GetMainWindow 函数来获取父类窗口，而不用 this 指针。该函数代码如下：

```
void TMyApp::CmAbout()
{ TDialog(GetMainWindow(),IDD_ABOUT)Execute();
}
```

五、EvDropFiles 函数

EvDropFiles 处理 WM_DROPFILES 文件，该函数带一个 TDropInfo 对象的参数。TDropInfo 对象含有函数，用来找出要放进窗口中的文件数目，文件名，及文件要放在哪儿等等。

由于本例是 SDI 应用程序，如果拖动的文件数大于1，必须警告用户一次只能将一个文件放进应用程序中。用 TDropInfo 的函数 DragQueryFileCount 来得到拖放的文件数目，该函数不带参数，返回要拖放的文件数。如果文件数大于1，弹出消息框通知用户。

下一个是获得拖放的文件名。调用 TDropInfo 的函数。DragQuery FileNameLeg 可以找出文件路径字符串的长度。该函数只带一个参数，即欲查询的文件的索引，在本例中，因为只有一个文件，所以该参数必须设为0，该函数返回文件路径的长度。

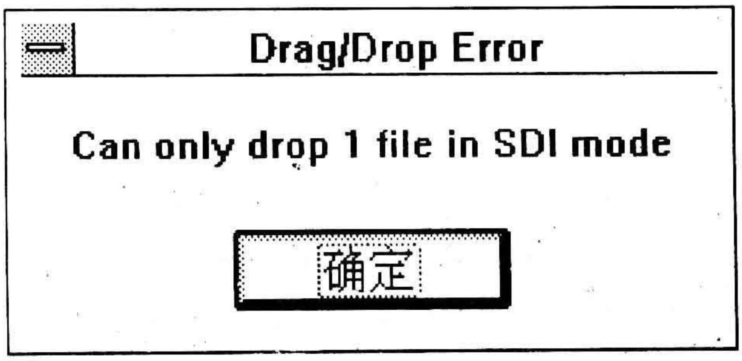
分配一个文件长度大小的字符块，然后调用 TDropInfo 函数 DragQueryFile。该函数带三个参数，第一个是文件的索引，其值必须为0(因为只有一个文件)；第二个是文件路径 Char *，第三个参数是文件路径的长度。函数用第2个参数中的字符串填充文件路径。

一旦定位了文件名，就必须获取恰当的文件类型模板；调用文档管理器的 MatchTemplate 函数可以做到这一点。该函数搜索文档管理器的文档模板链表；并将匹配拖放文件模式的第一个文档模板指针返回；该指针是 TDocTemplate *。如果文档管理器找不到匹配的模板，则返回0。

一旦定位了模板，就可以调用模板的 CreateDoc 函数，该函数带文件路径参数。该函数创建一个新文档及相关的浏览器，并在文档中打开该文件。

EvDropFiles 函数如下：

```
void
TMyApp::EvDropFiles(TDropInfo dropInfo)
{
    if(dropInfo.DragQueryFileCount()!=1)
        ::MessageBox(0,"Can only drop 1 file in SDI mode","Drag/Drop Error",MB_OK);
    else{
        int fileLength=dropInfo.DragQueryFileNameLen(0)+1;
        char * filePath=new char [fileLength];
        dropInfo.DragQueryFile(0,filePath,fileLength);
        TDocTemplate * tpl=GetDocManager()-> MatchTemplate(filePath);
        if(tpl)
            tpl->CreateDoc(filePath);
        delete filePath;
    }
    dropInfo.Dragfinish();
}
```



六、EvNewView 函数

当和浏览器相关的事件产生时，WM_OWLVIEW 事件就通知应用程序。处理 WM_OWLVIEW 事件的函数带一个 TView& 参数并返回 void。如果事件的参数是 dnCreate，表示要创建一个新浏览器，通知应用程序设置浏览器窗口。

本例中，将浏览器窗口作为主窗口的客户窗口，要调用 GetWindow 和 SetClientWindow

两个函数。

GetClientWindow 是 view 类的成员。该函数没有参数，返回 TWindow *，指向浏览器窗口。

一旦拥有了指向浏览器窗口的指针，就可用主窗口的函数 SetClientWindow 将该浏览器窗口设为客户窗口。该函数带一个 TWinlow * 参数，且返回一个 TWindow * 。这个指针是指向先前客户窗口的指针（如果先前有客户窗口的话）。

必须检测这个新的客户窗口是否成功地创建了。TView 提供 IsOK 函数。如果该函数返回 FALSE，说明窗口创建失败，此时就要再次调用 SetClientWindow，并将参数设为0，然后从该函数返回。

如果窗口创建成功，就须用 View 的 GetViewMenu 函数检测浏览器的菜单。如果 View 有菜单，就要用主窗口的 MergeMenu 函数将 View 的菜单和窗口的菜单组合起来。

EvNewView 的代码如下：

```
void
TMyApp::EvNewView(TView& view)
{
    GetMainWindow()->SetClientWindow(view,GetWindow());
    if(! view.IsOK())
        GetMainWindow()->SetClientWindow(0);
    else if (view.GetViewMenu())
        GetMainWindow()->MergeMenu( * view.GetViewMenu());
}
```

七、EvCloseView 函数

如果 WM_OVLVIEW 事件的参数为 dnClose，说明浏览器已经关闭。如同 EvNewView 函数一样，EvCloseView 函数返回 void，其参数为 TView&。

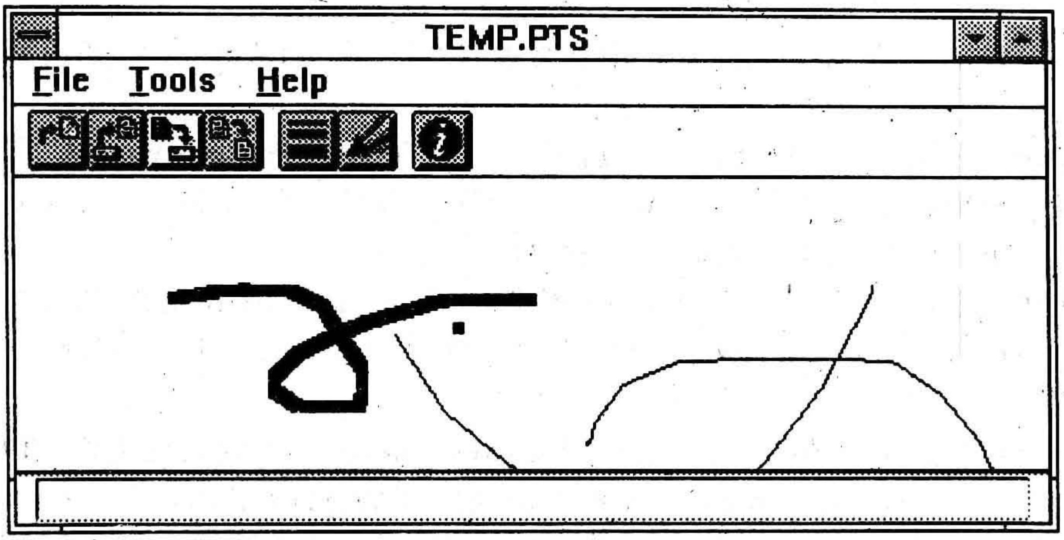
要关闭一个浏览器，必须将作为主窗口的客户窗口的浏览器窗口移去。调用主窗口的 SetClientWindow 函数，并传参数0就可达到目的，然后调用 RestoreMenu 函数将框架窗口的菜单恢复成原状。

当用 EvNewView 函数创建浏览器时，框架窗口的标题设为文框的文件路径。必须用 SetCaption 将主窗口标题恢复。

EvCloseView 的代码如下：

```
void
TMyAPp::EvCloseView(TView&/ * view * /)
{
    GetMainWindow()->SetCLientWindow(0);
    GetMainWindow()->RestoreMenu();
    GetMainWindow()->SetCaption("Drawing Pad");
}
```

第十一步运行结果如下图所示：



5.12 第十二步:讨论 MDI

如果 Doc/View 模块用在多文框接口(MDI)应用程序中,就会更有用。一个框架拥有多个子窗口,就可以给一个文档打开多个浏览器。

在十二步中,将给应用程序增加 MDI 的功能,需要在 TDrawDocument 和 TDrawView 类中增加新函数。另外,还要增加新特征:如删除直线,修改一条直线;undo 功能等。另还要创建一个新类 TDrawListView 来显示多浏览器窗口。TDrawListView 是显示存在 TDrawDocument 中的图形的一个可选浏览器。

5.12.1 在应用程序中支持 MDI

STEP12.CPP 包括应用程序对象和定义主窗口的代码。应用程序对象为定义在 STEP12DV.CPP 的 Doc/View 类提供整体框架。这一部分介绍 TMyAPP 类的一些修改,以便为 Doc/View 应用程序提供 MDI 支持。主函数 OWLMain 没有修改。

一、转化成 decoratedMDI 框架

为支持 MDI 应用程序,必须把 TDecoratedFrame 改为 TDecoratedMDIFrame。接着,在 decoratedMDI 框架中,创建一个 MDI 客户窗口(用类 TMDIClient)为便于在后续程序中定位客户窗口,在 TMyApp 类中增加一个 TMDIClient * 成员 Client。这个 MDI 客户窗口包含 MDI 子窗口,用来显示不同的浏览器。TDecoratedMDIFrame 的构造函数和 TDecoratedFrame 的构造函数不同。TDecoratedMDI Frame 的构造函数带5个参数。后三个参数都有缺省值,编程时不用提供值,其构造函数如下:

```
TDecoratedMDIFrame(const char far *  title,
                   TResId            menuResId,
                   TMDIClient&       clientWnd= * new TMDIClient,
                   BOOL              trackMenuSelection=FALSE,
                   TModule *         module=0);
```

这里：

- title 是框架窗口标题；
- menuResId 是窗口菜单的菜单资源标识符；
- clientWnd 是 TMDIclient 窗口对象的引用；
- trackMenuSelection 规定是否跟踪菜单命令；
- module=0，不同。

窗口标题 title 为"Drawing Pad"。该窗口没有菜单资源，用 TMenu Descr 来构造菜单。需要创建客户窗口并将它赋给数据成员 Client。最后，设菜单跟踪为 ON，窗口构造函数如下：

```
TDecoratedMDIFrame * frame=new TDecoratedMDIFrame
("Drawing Pad",0, * (client=new TMDIClient),TRUE);
```

二、改变提示模式

在11步，用户可能注意到，只有按下按钮后，控制条的提示文本才会出现，可以改变这种提示模式，只要鼠标移到按钮上，就显示提示文本。

调用控制条的 SetHintMode 函数，并将参数设为 TGadgetWindow::EnterHints，如下：

```
cb-> SetHintMode(TGadgetWindow::EnterHints);
```

三、修改拖放函数

EvDropFiles 处理 WM_DROPFILES 事件，该函数带五个 TDropInfo 对象参数，TDropInfo 含有要拖进窗口中的文件数目、文件名、及文件放在何处等信息。

本例和第11步不同它是 MDI 程序，因而拖的文件数目可以大于1。首先用 TDropInfo 的函数 DragQuery FileCount 来获取拖放文件的数目；该函数不带参数，返回拖动文件的数目；下一步是获得每个文件的文件名。调用 TDropInfo 的函数 DragQueryFileNameLen 可以找出文件路径的长度。该函数带一个参数，即是文件的索引值；函数返回文件路径的长度。

然后分配一个文件长度大小的字符块；调用 TDropInfo 的函数 DragQueryFIle，该函数带三个参数，叙述同第11步相关部分。

一旦获得了文件名，就要获取恰当的文件类型模板。调用文档管理器的 MatchTemplate 函数完成这一点，具体做法同第11步中类似部分。

一旦定位了模板，就要调用模板的 CreateDoc 函数，该函数带文件路径参数，它创建新文档及其相关的浏览器，并在文档中打开该文件。

所有文件都打开后，调用 DragFinish 函数。该函数释放在拖放操作过程中 Window 分配的内容。

EvDropFiles 函数代码如下：

```
void
TMyApp::EvDropFiles(TDropInfo dropInfo)
{
    int fileCount = dropInfo.DragQueryFileCount();
    for(int index=0;index<fileCount;index++){
```

```
            int fileLength=dropInfo.DragQueryFileNameLen(index)+1;
            char * filePath=newchar [fileLength];
            dropInfo.DragQueryFile(index,filePath,fileLength);
            TDocTemplate * tpl=GetDocManager()->MatchTemplate(filePath);
            if(tpl)
                tpl->CreateDoc(filePath);
            delete filePath;
        }
        dropInfo.Dragfinish();
    }
```

四、关闭浏览器

在Step11中,要关闭一个浏览器,得将作为客户窗口的浏览器移走,恢复主窗口菜单,重设主窗口标题。在MDI应用程序中,这些都不必再做,MDI窗口类已经为你做了这一切。

EvClose 函数代码如下:

```
TMyApp::EvCloseViewTView&/ * view * /)
{ // 在 MDI 中,不用做任何事
}
```

5.12.2 修改 TDrawDocument 和 TDrawView 类

需要在 TDrawDocument 和 TDrawView 类中做以下改动,这些改动包括:定义新事件,增加新事件处理函数,增加文档特性函数等。

一、定义新事件

首先要定义三个新事件,以支持 TDrawdocument 和 TDrawView 类中的新特点。这些浏览器的通知消息是 vnDrawAppend,vnDrawDelete 和 vnDrawModify。这些事件必须为 const int,并在予定义值 vnCustomBase 上增加编移量。使用 vnCustomBase 保证用户定义的事件不会覆盖 objectWindows 事件。

接下来,用 NOTIFY-SIG 宏定义事件响应函数,这个宏带两个参数,事件名(如 unDrawAppend 或 unDrawDelete)和传给事件处理函数的参数类型。参数类型的大小不要超过 long;如果传递的对象大于 long 必须用指针传递。本例中,参数仅是 unsignedint 将直线的索引传给事件处理函数。事件处理函数的舞台值为 void。

最后,为每个事件定义响应表。为简便起见,宏名就用事件名,全部用大写,打头为 EV—VN_。用#define 宏来定义宏名。

宏 VN_DEFINE 的语法如下:

```
VN_DEFINE(eventName,functionName,paramSize);
```

这里:

- euentName 是事件名;

■ functionName 是事件处理函数名；

■ paramSize 是传给事件处理函数的参数大小，可以是下列四个值之一：

- void
- int（整型值，依赖于平台）；
- long（32位整数或 far 指针）；
- 指针 Dointer（大小依赖于内存模式）。

必须选择最接近于事件处理函数的参数类型大小。定义新事件的代码如下：

```
const int vnDrawAppend=vnCustomBase+0;
const int vnDrawDelete=vnCustomBase+1;
const int vnDrawModify=vnCustomBase+2;

NOTIFY_SIG(vnDrawAppend,unsingned int)
NOTIFY_SIG(vnDrawdelete,unsigned int)
NOTIFY)SIG(vnDrawModify,unsigned int)

#define EV_VN_DRAWAPPEND VN_DEFINE(vnDrawAppend,VnAppend,int)
#define EV_VN_DRAWDELETE VN_DEFINE(vnDrawDelete,VnDelete,int)
#define EV_VN_DRAWMODIFY VN_DEFINE(vnDrawModify,VnModify,int)
```

二、修改 TDrawDocument

TDrawDocument 增加几个新的保护数据成员：

■ UndoLine 是一个 TLine * 。用于存储 Linews 数组中线被修改或删除后，原始的直线，以便恢复。

■ UndoState 是 int 型，用来指示最后一次用户操作的性质，反向做最后一次操作就可完成 undo 可以是下列四值之一：

- UndoNone 表示没有操作可供 Undo；
- UndoDelete 表明从文档中删除了一条线；
- UndoAppend 表明向文件中增加了一条线；
- UndoModify 表明修改了文档中的一条线；

■ UndoIndex 是 int 型，它包含最后一条被修改线的索引，以便修改工作可以 Undo。

■ FileInfo 是字符串，它含有关文件的信息，这个字符串和 Step11 中 TDrawDocument::Commit 函数存储的文件信息相同，TDrawDocument 构造函数应将 UndoLine 初始化为0，将 UndoStute 初始化为 UndoNone。~TDrawDocument 删除 UndoLine。

稍微修改 Open 函数，以从文档文件读了文件信息，来初始化 FileInfo 成员。如果文档路径不合法，用字符串资源 IDS_FILEINFO 初始化 FileInfo。

修改 AddLine 函数，当在图形中增加一条线后，通知所有的浏览器，可以用带 unDrawAppend 事件的 NotifyViews 函数。NotifyView 的第二个参数必须是新的直线数组的索引。也要将 undoState 设为 UndoAppend Addline 函数代码如下：

```
int
TDrawDocument::AddLine(TLine& line)
{
    int index=Lines->GetItemsInContianer();
    Lines->Add(line);
    SetDirty(TRUE);
    NotifyViews(vnDrawAppend,index);
    UndoState=UndoAppend;
    return index;
}
```

三、属性函数

每个文档都有一个属性列表。每个属性有一个相关值，定义为 enum，通过它来区分。派生类的列表必须以 NextProperty 结束，以 PrevProperty 开始，且将该值设为基类的 nextProperty 成员减去1。

每个属性还有一个文本串和一个 int 值。文本串存在一个叫 PropNames 的数组中，用来描述属性；int 值存在一个叫 PropFlags 的数组中。

属性的 enum 值用在数组索引中，以定位属性字符串或具体属性的标志。

TDrawDocument 将两个属性加在文档的属性列表中：LineCount 和 Description。Enum 定义如下：

```
enum{
PrevPropety=TFileDoucment::NextProperty-1,
Line Count,
Description,
nextProperty,
};
```

通过重新定义 PreProperty 和 NextProperty，任何从用户定义文档派生出的类都可以创建新属性而不用重写刚定义的属性。

TDrawDocument 也增加了一个 Static char 字符串数组。这个数组包含两个字符串，描述新属性，数组定义如下：

```
Static char * PropNames[]={
"Line Count",
"Description",
};
```

最后，TDrawDocument 增加一个 PropFlags 静态整型数组，它含有和 PropNames 相当的项数。每个数组元素含有一个或多个用 OR 组成的属性标志，并和 PropNames 中的属性有相同的数组索引。

PropFlags 数组定义如下：

```
Static int PropFlags[]={
```

```
pfGetBinary|pfGetText,    // LineCount
pfGetText                 // Description
};
```

TDrawDocument 重载了许多 TDocument 的属性议程来提供获取新属性的功能。通过调用 PropertyCount 函数来获得属性的数目。PropertyCount 返回的属性值为 enum 量 NextPropery 减1。

调用 ProperName 函数来获取任何文档属性的文本名。Propertyname 返回含有属性名的 Char * 指针，该函数带一个 int 参数，这个参数表示俗查找的文本名的索引，如果索引值小于或等于 PrevProperty，就可以调用 TFileDocument::PropertyName；它返回定义在 TFileDocument 或其基类 TDocument 中的属性名。如果索引值大于或等于 nextProperty，就应该返回0。NextProperty 标识了文档类中的最后一个属性。如果索引值大于或等于 NextProperty，该值就太大了，因而不合法。只在索引值在 PrevProperty 和 NextProperty 之间，就返回 Propnames 数组中索引指向的字符串 PropertyName。代码如下：

```
const char *
TDrawdocument::PropertyName(int index)
{
    if (index<=PrevProperty)
        return TFileDocument::PropertyName(index);
    else if(index<NextProperty)
        return PropNames[index-PrevProperty-1];
    else
        return 0;
}
```

函数 FindProperty 本质上和 Property Name 函数功能相反，FindProperty 带一个 Const Char * 参数。该函数将传来的字符串和每一个文档属性匹配，如果字符串正好和属性名匹配，它返回该属性的索引值，FindProperty 函数如下：

```
int
TDrawDocument::FindProperty(const char far * name)
{
    for (int i=0;i<NextProperty-PrevProperty-1;i++)
        if(strcmp(PropNames[i],name)==0
            return i+PrevProperty+1;
        return 0;
    }
```

PropertyFlags 函数带一个整型参数，它是属性标志的索引。这些标志作为一个 int 型返回。如果该索引小于或等于 PrevProperty，就可以调用 TFileDocument::PropertyName，它返回定义在 TFileDocument 或其基类 TDocument 中的属性名。如果索引值大于或等于 NextProperty。就返回0，NextProperty 是文档类的最后一个属性。如果属性值大于或等于 NextProperty，该索引值就不合法。只要索引值在 PrevProperty 和 NextProperty 之间就返回

PropFlags 数组中的相应的索引值。PropertyFlags 函数代码如下：

```
int
TDrawDocumetn::PropertyFlags(int index)
{
    if (index <= PrevProperty)
        return TFileDocument::PropertyFlags(index);
    else if(index <NextProperty)
        return PropFlags[index-PrevProperty-1];
    else
        return 0;
}
```

最后一个属性函数是 GetProperty，它带三个函数。第一个参数是 int 型，是属性的索引，第二个参数是 void *，它用来存属性信息，是一块内存，第三个参数是这块内存的字节数。

GetProperty 必须处理三种可能性：

- LineCount 属性有文本和二进制两种方式。要以二进制形式获取 LineCount 属性，须将 GetProperty 的第三个参数设为0，这时第二个参数应该指向一个合适的数据对象来存适当的数据。若要以文本方式获取 LineCount 属性，须将 GetProperty 函数的第二个参数指向一个合法的内存块；第三个参数设为这个内存块的大小。
- DeScription 属性只能以文本方式获取。只要将 FileInfo 字符串拷入第二个参数中。
- 如果文本属性既不是 LineCount，也不是 Description，则调用 TFileDocument::GetProperty 函数。

GetProperty 函数代码如下：

```
int
TDrawDocumetn::GetProperty(int prop,void far * dest,int textlen)
{
    swithc(prop)
    {
        case LineCount:
        {
            int count=Lines->GetItemsInContainer();
            if( ! textlen){
                * (int far * )dest=count;
                return sizeof(int);
            }
            return wsprintf((char far * )dest,"%d",count);
        }
        case Description:
            char * temp=new char[textlen];// need local copy for medium model
            int len=FileInfo.copy(temp,textlen);
            strcpy((char far * )dest,temp);
```

```
            return len;
        }
        return TFileDocumetn::GetProperty(prop,dest,textlen);
    }
```

四、TDrawDocument 中新增函数

TDrawDocument 新增了几个函数。这些函数用来修改文档对象。可以用来删除线，修改线，清除文档和取消对文档的修改等。

第一个新增函数是 DeleteLine。这个函数的功能是删除文档中的一条线。该函数带一个 int 参数，是欲删除的直线的数组索引。

- DeleteLine 必须检测传来的索引值是否合法，这可以通过调用 GetLine 函数并将索引值传给 GetLine。如果索引值是合法的，GetLine 返回一个指向直线对象的指针，否则返回0。
- 一旦肯定索引值是有效的，就把 UndoLine 设为要删除的那条线，并把 UndoSetate 设为 UndoDelete。保存删除线以防用户要求取消(undo)删除动作。
- 调用容器类函数 Detach 将删除线从文档中断开，该函数带一个 int 参数，是欲删除线的数组索引。
- 调用 SetDirty 函数将 Isdirty 标志设为"on"
- 调用 NotifyViews 函数通知浏览器文档已经改变；将 unDrawDelete 作为第一个参数传给 NotifyViews，第2个参数是直线的数组索引。DeleteLine 函数代码如下：

```
void TDrawdocument::DeleteLine(unsigned int index)
{
    const TLine * oldLine=GetLine(index);
    if(!oldLine)
        return;
    delete UndoLine;
    UndoLine=new TLine( * oldLine);
    Lines->Detach(index);
    SetDirty(TRUE);
    NotifyViews(vnDrawDelete,index);
    UndoState=UndoDelete;
}
```

ModifyLine 函数带两个参数：一个 TLine& 和一个 int。int 是欲修改的直线的数组索引，被修改的线由 TLine& 代替。

- 在替换一条线之前，要设置 undo 数据成员，将欲删除的线拷入 UndoLine，并设 UndoState 为 UndoModify，同时还要将 UndoIndex 设为受影响的那条线的索引。
- 将直线对象设为传进来的 TLine 对象。
- 调用 SetDirty 将 IsDirty 设为"on"
- 最后，为调用 NutifyViews 通知浏览器文档已经修改。将 VnDrawModify 作为第一个

参数传给 NotifyViews，数组索引作为它的第2个参数。

Modify Line 函数代码如下：

```
void TDrawdocument::ModifyLine(TLine& line,unsigned int index)
{
    delete UndoLine;
    UndoLine=new TLine((*Lines)[index]);
    SetDirty(TRUE);
    (*Lines)[index]=line;
    NotifyViews(vndrawModify,index);
    UndoState=UndoModify;
    UndoIndex=index;
}
```

Clear 函数简单明了，它刷删新 Lines 指向的 TLines 数组，然后强迫浏览器更新，通过调用 NotifyViews 并传参数 vnRevert，浏览器更新后，由于文档中没有数据，浏览器便清除自己的窗口。Clear 函数如下：

```
void TDrawDocument::Clear();
{ Lines->Flush();
    NotifyViews(vnRevert,TRUE)
}
```

Undo 函数有三种不同种类的 undo 操作：

Append、Delete 和 Modify。通过 UndoState 变量决定需要进行哪种 Undo 操作。

- 如果 UndoState 为 UndoAppend，Undo 删除加入数组中中的最后一条线；
- 如果 UndoState 为 UndoDelete，Undo 将把 UndoLine 指向的线加入数组中；
- 如果 UndoState 为 UndoModify，Undo 恢复由 UndoLine 指向的线到数组中特定位置，这个位置由 UndoIndex 指定。

Undo 函数的代码如下：

```
void TDrawdocument::Undo()
{
    switch(UndoState){
        case UndoAppend:
            DeleteLine(lines->GetItemsIncontainer()-1);
            return;
        case UndoDelete:
            AddLine(*UndoLine);
            delete UndoLine;
            UndoLine=0;
            return;
        case UndoModify:
```

```
            TLine * temp=UndoLine;
            UndoLine=0;
            ModifyLine( * temp=UndoIndex);
            delete temp:
        }
    }
```

五、TDrawView

TDrawView 修改了几个函数,包括删除 GetPenSize 函数。这个函数放到 TLine 类中,这样笔的尺寸由线本身设置,在 CmPenSize 函数中调用 TLine::GetPenSize。同样,将 GetPen-Color 移到 TLine 类中,在 CmPenColor 中调用 TLine::GetPenColor 函数。

为在 TDrawDocument 和 TDrawView 类中增加新的编辑功能,需要为 Undo 和 Clear 增加菜单选项,这些选项处理 CM_DLEAR 和 CM_UNDO。在 SetViewMenu 调用中,需要修改 TMenuDescr 构造函数参数。新函数如下:

```
SetViewMenu(new TMenuDesct(IDM_DRAWVIEW,0,1,1,0,0,0);
```

也可以修改 EvRButtonDown 中重新定义鼠标右键行为(现在有两种方法改变画笔尺寸:一种是 Tools|Pen Size 菜单命令;另一种是控制条按钮)。定义 moust 右键为 undo 操作的热键,EvRButtonDown 函数如下:

```
void TDrawView::EvRButtonDown(UINT,TPoint &)
{ CmUndo();
}
```

六、TDrawView 中新增函数

第12步在 TDrawDocument 中增加了一些函数来完善与 TDrawDocument 的接口,需要重载 TView 中的虚拟函数 GetViewName,文档管理器调用该函数来决定浏览器类型。该函数返回指向浏览器名的 const Char * 指针,函数如下:

```
const char far * GetViewName()
{ return StateName(); }
```

在 Edit 菜单中增加 Clear 和 Undo 功能之后,必须处理 CM_CLEAR 和 CM_UNDO 事件。将下列行加入响应表中:

```
EV_COMMAND(CM_CLEAR,CmCLear),
EV_COMMAND(CM_UNDO,CmUndo);
```

定义响应两种事件的处理函数,如果浏览器接收到 CM_CLEAR 消息,只须调用文档的 Clear 函数:

```
void TDrawView::CmClear()
{ DrawDoc->Clear(); }
```

浏览器还须处理通知事件,vnDrawAppend,vnDrawDelete,vnDrawModify。须得这些事

件加到浏览器的响应表中：

```
DEFINE_RESPONSE_TABLE1(TDrawView,TWindowView)
    ⋮
    EV_VN_DRAWAPPEND,
    EV_VN+DRAWDELETE,
    EV_VN_DRAWMODIFY,
    ⋮
END_RESPNSE_TABLE;
```

处理这些事件的函数是 VnAppend,VnDelete,VnModify。这三种函数只带一个索引参数,即事件影响的是文档中的哪条线,函数返回 BOOL。

VnAppend 函数获取一个通知,一条线已被附加进文档中。然后,它在浏览器窗口中画出这些条新线,这时必须产生一个设备上下文,从文档中获取线,调用直线的 Draw 函数,该函数必须以设备上下文对象为参数,最后,函数返回 TRUE。

VnAppend 函数代码如下：

```
BOOL TDrawView::VnAppend(unsigned int index)
{
    TClientDC dc( * this);
    const TLine * line=DrawDoc->GetLine(index);
    line->Draw(dc);
    return TRUE;
}
```

VnModify 函数强迫整个窗口进行重绘,如果仅仅重画受到影响的窗口可能会更有效。但必须在已经存在的旧线上,重绘新加的线,然后再恢复覆盖了新画线的那些旧线。实际上将整个窗口变成无效,然后重绘,会更有效。所以 VnModify 的函数代码如下：

```
BOOL TDrawView::VnModify(unsigned int/ *  index * /)
{ Invalidate();  // 整个重绘
  return FRUE;
}
```

VnDelete 函数也迫使重绘整个窗口,这个函数和 VnModify 面临的问题相同,若简单地删除线可能会影响其他的直线。

VnDelete 的代码如下：

```
BOOL TDrawView::VnDelete(UNsigned int/ * index * /)
{ Invalidate();  // 整个重绘
  return;
}
```

5.12.3 TDrawListView

TDrawListView 类的目的就是将 TDrawDocument 对象中的数据作为一个直线链显示。每条线将显示自己的颜色,画笔大小,以及组成该线的点数。TDrawListView 允许改变线的

颜色和画笔大小。用户也可以删除一条线。

TDrawListView 从 TView 和 TListBox 派生出来，TViews 给 TDrawListView 标准的浏览功能，TListBox 则有能力将文档中信息显示在列表中。

一、创建 TDrawListView 类

TDrawListView 带两个参数，一个是 TDrawDocument 引用，另一个是指向父窗口的 TWindow * 指针，如果不提供值，父窗口缺省为0，构造函数将第一个参数传给 TView 构造函数并初始代 DrawDoc 成员，使其指向作为第一个参数传过来的文档。

TDrawListView 有两个数据成员，一个保护性的 TDrawdocument 指针的 DrawDoc，另一个为公有(public)成员 int CurIndex。DrawDoc 在 TDrawsListView 的目的和在 TDrawView 中的目的一样，即指向与浏览器相关连的文档对象。CurIndex 是列表框中当前选中的直线的数组索引。

TDrawListView 构造函数也调用 TListBox 的构造函数。TListBox 构造函数的第一个参数是由 TDrawListView 构造函数传过来的父窗口，第二个参数调用 TView 的函数 GetNextViewId，该函数返回 Static unsigned 用作列表框标识符。浏览器标识符在 TView 构造函数中设置。列表框的坐标和尺寸均设为0，在 MDI 子窗口中，将 TDrawListView 设为客户时，列表框的尺寸才会确定。

在构造函数中还设置一些窗口属性，在 AttrStyle 中，将 WS_BORDER 和 LBS_SORT 属性关闭；Attr. AccelTable 设为 IDA_DRAWLISTVIEW 加速键资源。

构造函数还设置菜单描述符。由于 TDrawListView 和 TDrawView 功能不同，菜单要求也不同。比较一下 TDrawView 和 TDrawList 和 TDrawListView 的菜单资源。

TDrawListView 的构造函数如下：

```
TDrawListView::TDrawListView(TDrawDocument& doc,TWindow, * parent)
        :TView(doc),TListBox(parent,GetNextViewID(),0,0,0,0),DrawDoc(&doc)
{
    Attr.Style &=(WS_BORDER |LBS_SORT);
    Attr.AccelTable=IDA_DRAWLISTVIEW;
    SetViewMenu(newTMenuDescr(IDM_DRAWLISTVIEW,0,1,0,0,0,0));
}
```

TDrawListView 没有动态分配的数据成员，所以折构函数不做任何事。

二、获取类名

像 TDrawView 类一样，TDrawListView 也应该定义函数 StaticName 来返回浏览器名。StaticName 函数如下：

```
static const char far * StaticName()
    { return "Draw List View"; }
```

三、重载 TView 和 TWindow 的虚拟函数

文档管理器调用浏览器函数 GetViewName 来决定浏览器的类型。必须重载这个函数，

该函数应该返回一个 const char * 它包含 view 名。GetViewName 如下：

```
const char far * GetViewName()
   { return StaticName(); }
```

文档管理器调用浏览器函数 GetWindow 获取和浏览器关联的窗口。此函数也要重载，并返回一个 TWindow * 。函数如下：

```
TWindow * GetWindow(){return (TWindow * )this;}
```

用户应该担任 CanClose 函数。这个函数必须调用 TListBox 的 CanClose 函数及文档的 CanClose 函数。该函数如下：

```
BOOL CanClose(){return TListBox::CanClose()&& Doc->CanClose(); }
```

用户还要提供 Create 函数，调用 TListBox 的 Create 来实现创建窗口，但必须将数据从文档调入 TDrawListView 对象，通过调用 LoadData 函数实用。Create 函数代码如下：

```
BOOL TDrawListView::Create()
{ TListBox::Create();
  LoadData();
  return TRUE;
}
```

四、载入和格式化数据

必须提供函数来装入数据。这些函数从文档对象将数据装入浏览器的文档并格式化这些数据，以便在列表框中显示。这些函数应该定义为 protected，只有浏览器能调用它们。

第一个函数是 LoadData。首先要清除列表框中可能有的列表项，再把数据调入列表框，要做到这一点，必须调用 TListBox 在类的 ClearList 函数，然后从文档中调入直线并格式化每条直线，直到文档中全部线调完为上。如果文档中再没有线，GetLine 函数返回0。最后，调 SetSelIndex 函数将当前选择索引设为0，LoadData 函数代码如下：

```
void TDrawListView::LoadData()
{
     ClearList();
     int i=0;
     const TLine * line;
     while((line=DrawDoc->GetLine(i))!=0)
          Formatdata(line,i++);

     SetSelIndex(0);
}
```

FormatData 函数带两个参数，第一个参数是 const Tline * ，指向欲修改或添加进列表框的直线；第二个参数是要修改的直线的索引，FormatData 函数代码如下：

```
void TDrawListView::FormatData(const TLine * line,int unsigned index)
{
```

```
        char buf[80];
        TColor Color(line->QueryColor());
        wsprintf(buf,"Color=R%d G%d B%d,Size=%d,Points=%d",
                color.Rd(),color.Green(),color.Blue(),
                line->QueryPenSize(),line->GetItemsInContainer());
        DeleteString(index);
        InsertString(buf,index);
        SetSelIndex(index);
    }
```

五、TDrawListView 中的事件处理

TDrawListView 的响应如下：

```
    DEFINE_RESPONSE_TABLE1)TDrawListView,TListBox)
        EV_COMMAND(CM_PENSIZE,CmPenSize),
        EV_COMMAND(CM_PENCOLOR,CmPenColor),
        EV_COMMAND(CM_CLEAR,CmClear),
        EV_COMMAND(CM_UNDO,CmUndo),
        EV_COMMAND(CM_DELETE,CmDelete),
        EV_VN_ISWINDOW,
        EV_VN_COMMIT,
        EV_VN_REVERT,
        EV_VN_DRAWAPPEND,
        EV_VN_DRAWDELETE,
        EV_VN_DRAWMODIFV,
    END_RESPONSE_TABLE;
```

该响应表和 TDrawView 的响应表在某些方面相似。这两个响应表共享几个事件，如 CM_PENSIZE,CM_PENCOLOR,vnDrawAppend,vnDrawmodify。

但每个浏览器也处理其他浏览器不处理的事件。这是因为各个浏览器功能各异所致。例如，TDrawView 处理 Mouse 事件，而 TDrawListView 则不处理。这是因为在文本列表框中处理 Mouse 事件没有意义。这些事件在 TDrawView 窗口中画一条线时使用。

TDrawListView 处理 CM_DELETE 事件，而 TDrawView 不处理该事件。这是因为，在 TDrawView 窗口中，用户没有办法知道哪条线要被删除。但在列表框中，这很容易做到，仅仅删除列表框中当前选择的那条线。

TDrawListView 也处理 vnIsWindow 事件，vnIsWindow 事件是 ObjectWindows 预定义事件，它请求浏览器的窗口是否和随事件传递过来的窗口是同一个窗口。

在 TDrawListView 类中 CmPensize 函数较复杂，比 TDrawView 类中 CmPenSize 复杂，这是因为，TDrawListView 类不用维护指向当前直线的指针，而 TDrawView 却要维护，必须获得列表框中当前选择的直线的索引，并从文档中获取该线，由于 GetLine 返回指向 const 对象的指针，所以必须备份欲修改的直线，调整备份，然后调文档的 ModifyLine 函数。

CmPenSize 函数如下：

```
void TDrawListView::CmPenSize()
{
    int index==GetSelIndex();
    const TLine * line=DrawDoc->GetLine(index);
    if(line){
        TLine * newline=new TLine(*line);
        if(newLine->GetPensize())
            DrawDoc->ModifyLine(*newLine,index);
        delete newline;
    }
}
```

这个函数有趣的部分来自 ModifyLine 调用。当用户用这个函数改变画笔尺寸时，浏览器中的笔的尺寸此时并没有改变，但当文档在 ModifyLine 调用中修改直线时，它将 vnDrawModify 事件播向所有的浏览器。

```
NotifyViews(vnDrawModify,index);
```

它通知文档所有浏览器直线已经修改了。浏览器然后调用它们的 VnModify 函数来更新显示。这样，在一个浏览器中所做的修改会在其他浏览器中反映出来。其他函数如 CmPenColor，CmDelete，CmUndo 等修改文档数据时道理相同。

CmPenColor 函数和 CmPenSize 函数几乎一样，区别在：CmPenColor 调用直线的 GetPenColor 函数，而 CmPensize 调用直线的 GetPensize 函数。

CmPenColor 函数如下：

```
void TDrawListView::CmPenColor()
{
    iont index=GetSellIndex();
    const TLine * line->Drawdoc->GetLine(index);
    if(line){
        TLine * newLine=new TLine(*line);
        if(newline->GetPenColor())
            DrawDoc->ModifyLine(*newLine,index);
        delete newline;
    }
}
```

CM_DELETE 事件表示用户想删除当前列表框中选择的直线。浏览器要调用文档的 DeleteLine 函数，传给参数是当前选择的直线的索引。CmDelete 函数如下：

```
void TDrawListView::CmDelete()
{ DrawDoc->DeleteLine(GetSelIndex()); }
```

还要为 TDrawListView 提供 CM_CLEAR 和 CM_UNDO 的处理函数。如果用户选择 Clear 菜单命令，浏览器接收 CM_CLEAR 消息，调用文档的 Clear 函数：

```
void TDrawList::CmClear()
```

```
{ DrawDoc->Clear(); }
```

如果用户选择 Undo 菜单命令，浏览器接收 CM_UNDO 消息，调用文档的 Undo 函数：

```
void TDrawListView::CmUndo()
{ DrawDoc->Undo(); }
```

这些函数和 TDrawView 中的同一函数相同这是因为这些操作依赖于 TDrawDocument 来实际完成对数据的修改。像 TDrawView 类，TDrawListView. 的 VnCommit 函数总是返回 TRUE。在较复杂的应用程序中，这些函数将增加文档的 cach 数据（缓存数据）。但在本例中，当画每条线时，数据被加进文档中。

VnRevert 函数调用 LoadData 函数将列表框中的数据返回给文档。VnRevert 函数如下：

```
BOOL TDrawListView::VnRevert(BOOL/ * clear * /
{ LoadData();
  return TRUE;
}
```

VnAppend 函数带一个 Unsigned int 参数，即欲添加直线的索引值。通过调用文档的 GetLine 函数从文档中获取一条新线，调用 FormatData 函数，该函数带直线的指针及直线的索引两个参数，格式化直线之后，将选择索引设为新加直线并返回。函数如下：

```
BOOL TDrawListView::VnAppend(unsigned int index)
{
    const TLine * line=DrawDoc->GetLine(index);
    FormatData(line,index);
    SetSelIndex(index);
    return TRUE;
}
```

VnDelete 函数带一个欲删除的直线索引，调用 ListBox 的函数 DeleteString 将线从列表框中删除。函数如下：

```
BOOL TDrawListView::VnDelete(unsigned int index)
{
    DeleteString(index);
    HandleMessage(WM_KEYDOWN,VK_DOWN);   // force selection
    return TRUE;
}
```

调用 HandleMessage 是为了保证当前线被删除之后，列表框中还有活动选项。

VnModify 函数带一个欲调整直线的索引参数，需要调用 GetLine 函数从文档中获取直线。调用带 line 和其索引的 FormatData 函数。

```
BOOL TDrawListView::VnModify(unsigned int index)
{
    const TLine * line=DrawDoc->GetLine(index);
    FormatData(line,index);
```

```
    return TRUE;
}
```

5.13 第十三步:学习编写DLL程序

动态链接库是Windows很重要的组成部分。由于动态链接库在内存中只有一个实例(拷贝)存在,因此,当多个应用程序共享同一个DLL时,可以使程序有效的使用内存。

在BC4.0中,应用程序有16位和32位运行代码之分。在编写16位的DLL时,要注意以下几点:

1. 调用16位的DLL函数时,必须是远程调用,函数的指针参数或者指针返回值都必须是远程指针。
2. 在16位DLL中定义的静态数据是全局数据。如果要使某个数据只被某个16位的DLL应用程序调用,必须地动态地创建和管理这个数据。

对于32位的DLL程序,函数调用是远程调用,指针也是远程指针;定义的静态数据也是私有数据而不是全局数据。

每个DLL程序都有一个入口函数出口函数,在BC4.0中对于16位的DLL程序,入口程序是LibMain函数,出口函数是WEP函数,对32位的DLL程序,入口和出口函数都是DL-LEntryPoint。

有关这些具体的函数说明以及注意事项,参考动态链接库一章。

下面以一个具体例子来说明,先列出本例的源程序。在入口函数LibMain中,去检测静态TModule指针是否为NULL,如果为NULL,说明还没有创建TModule,DLL还没有调入内存,于是调用TModule构造函数构造一个TModule对象,如果创建成功,状态返回值为零,否则状态返回值为非零,删除该对象,函数返回0。

在出口函数中一般做一些清理工作。在这里要特别指出,在编译DLL程序时,使用的是Large存储模型,在DLL中用ntw创建的对象(如对话框)的所有权不属于该动态链接库,而属于调用该动态链接库的第一个应用程序,当应用程序终止时,该对象也随着消失,因此在WEP函数中,不能再使用delete DLLDialogLib。

下面给出使用这个动态链接库的例子。

```
TTestWindow::CmCreate()
{ CreateDLLDialog(HWindow);
}
```

在创建动态对话框时,将父窗口句柄传给创建函数。当用户选择Create菜单时,弹出一个对话框。

动态链接库程序如下:

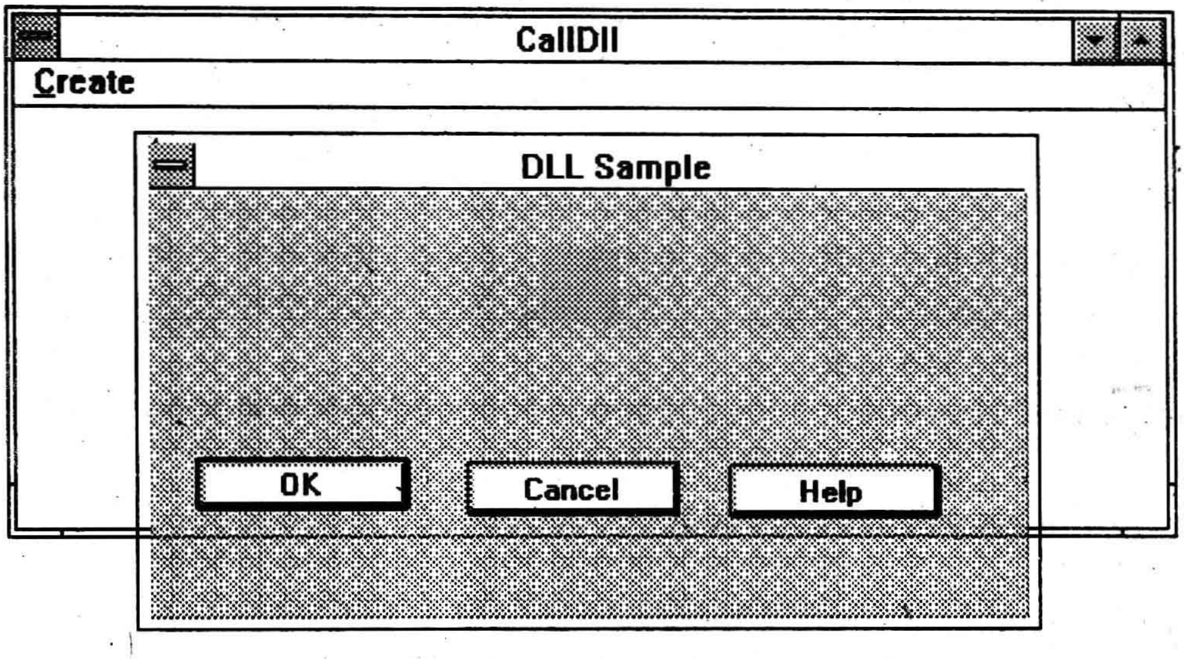
```
#include<owl\owlpch.h>
#include<owl\applicat.h>
#include<owl\dialog.h>
#include<owl\window.h>
```

```
#include "dlldlg.h"
#include <stdlib.h>
static  TModule  *DLLDialogLib;
BOOL far _export
CreateDLLDialog(HWND parentHWnd)
{
    TWindow *  parentAlias=GetWindowPtr(parentHWnd);   // check if an OWL iwnod
    if(!parentAlias)
        parentAlias-=new TWindow(parentHWnd);
    TDialog  *dlg=new TDialog (parentAlias,TResId(1),DLLDialogLib);
    return (dlg->Execute());
}
int FAR PASCAL LibMain(HINSTANCE hInstance,WORD/* wDataSeg */,
    WORD   /* cbHeapsize */,LPSTR /* lpCmdLine */)
{
        int TheStatus;
        // Note that we can't ensure that DLLDialogLib is allocated
        // in memory owned by DLLHELL0.DLL. Hence,it will get owned
        // by the first application that caused DLLHELL0.DLL to be loaded.
        if(!DLLDialogLib)
            DLLDialogLib=new TModule(0,hInstance);
            THeStatus=DLLDialogLib->Status;
            if(TheStatus!=0)
            {
            delete DLLDialogLib;
            DLLDialogLib=NULL;
        }
        return (TheStatus==0);
}
int FAR PASCAL WEP(int/* bSystemExit */)
{
    // don't delete DLLDialogLib here. We haven't guaranteed that
    // DLLLDialogLib points to memory owned by DLLHELL0.DLL,It
    // Will belong to the application that first caused DLLHELL0.DLL
        // to be loaded. At the time WEP is called,that application,
    // and all of its memory,has been deleted.
        reutrn 1;
}
```

测试程序如下：

```
#include<owl\owlpch.h>
#include<owl\applicat.h>
#include<owl\framewin.h>
```

```
#include<owl\dc.h>
#include "Dlldlg.h"
#include <cstring.h>

#define CM_CREATE 101

class TTestWindow:public TWindow{
    public:
        TTestWindow():TWindow(0,0,0){}
        void CmCreate();   // Call into a DLL to create a window.
        DECLARE_RESPONSE_TABLE(TTestWindow);
};

DEFINE_RESPONSE_TABLE1(TTestWindow,TWindow)
    EV_COMMAND(CM_CREATE,CmCreate),
END_RESPONSE_TABLE;

//  Call into DLL to create window.
//
void
TTestWindow::CmCreate()
{
    CreateDLLDialog(HWindow);
}

class TCallDllaApp:public TApplication{
    public:
        void InitMainWindow();
};

void
TCallDllApp::initMainWindow()
{
    MainWindow=new TFrameWindow(0,"CallDll",new TTestWindow);
    MainWindow->AssignMenu("COMMANDS");
}

int
Owlmain(int/ * argc * /,char * / * argv * /[])
{
    int ret=TCallDllApp() Run();
    return ret;
}
```



5.14　第十四步:在 OWL 程序中使用自己编写的双链表

Borland C＋＋ 4.0自己附带一套类库和容器库,对常见数据结构如数组,单向链表,双向链表,B 树等有它一套较成熟的函数;但由于其复杂程度很高,一般用户很难运用自如。一些 DOS 的 C,C＋＋用户,曾经编写过数据结构方面的程序,现在想利用现有的自己很熟悉的程序,将它移植到 OWL 程序中,如何写这方面的程序呢?下面以双向链表为例来说明。

双向链表是很常见的一种数据结构,对它可以做添加,删除,修改,插入等操作。双向链表中的节点数据是什么类型,我们并不需要关心。这些节点数据从一个共同的基类派生出来。首先定义一个节点数据的基类,如下:

```
class TDatObject
{ public:
        class TDataObject  * next;
        class TdataObject  * Prior;
        // default constructor
        TDataObject()
        { Next=NULL;
          Prior=NULL;
        }
        virtual TDataObject  * GetPrior()
        { return Prior;
        }
        virtual TDataBOject  * GetNext()
        { return Next;
        }
        virtual TDataObject  * CreateNode()
```

```
        { return NULL;}
        virtual void GetValue(char *value1,char *value2,char *value3){}
        virtual void GetName(char *name)
        {
        }
};
```

在实际中,我们并不使用这个基类,这个基类只是封装了派生类的一些共同行为,所有标识为 virtual 的函数在派生类中都可以重载。函数 CreateNode()用来创建一个节点数据,在派生类中必须重写这个函数。

用户可以从 TDataObject 中派生出适合自己专用的节点数据例如派生一个 TRectObject,如下:

```
class TRectObject:public TDataObject
{ public:
    char Name[20],Value1[40],Value2[40],Vlue3[40];
    TRectObject(char *name,char *v1,char *v2,char *v3)
    :TDataObject()
    { strcpy(Name,name);
      strcpy(Value1,v1);
      strcp(Value2,v2);
      strcpy(Value3,v3);
    }
    virtual TDataObject *CreateNode()
    { return new TRectObject(name,Value1,Value2,Value3);
    }
    virutal void GetValue(char *value1,char *value2,char *value3)
    {
      strcpy(value1,Value1);
      strcpy(value2,Value2);
      strcpy(value3,Value3);
    }
    virtual void GetName(char *name)
    { strcpy(name,Name);
}
```

在这个派生类中,我们重载了基类中的函数 CreateNode,这个函数在这个派生类中创建个 TRectObject 型对象作为节点数据。

用户还可以继续从基类 TDataObject 派生其他子类,在这个子类中,定义自己的专用的私有(或公有)数据成员,重写 CreateNode 函数等一些 virtual 函数。

在定义了双链表中节点数据的类等级之后,我们要定义操作这些节点数据的双向链表,定义一个链表如下:

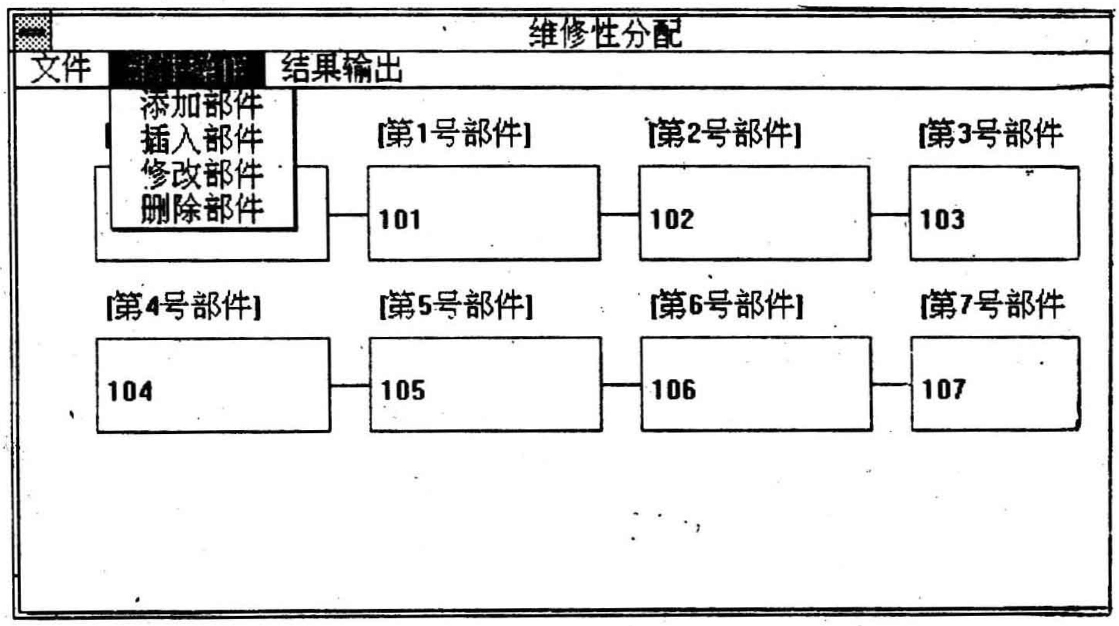
```
class TDoubleList
```

原书缺页

原书缺页

原书缺页

原书缺页


维修性分配
文件
结果输出
添加部件
插入部件
修改部件
删除部件
[第1号部件]
[第2号部件]
[第3号部件
101
102
103
[第4号部件]
[第5号部件]
[第6号部件]
[第7号部件
104
105
106
107

Document generated by Anna’s Archive around 2023-2024 as part of the DuXiu collection (https://annas-blog.org/duxiu-exclusive.html).

Images have been losslessly embedded. Information about the original file can be found in PDF attachments. Some stats (more in the PDF attachments):
{
  "filename": "44CKQm9ybGFuZCBDKysgNC4wIGZvciBXaW5kb3dz6Z2i5ZCR5a+56LGh56iL5bqP6K6+6K6hICBPV0wyLjAgZm9yIEMrK+OAi18xMjc2NjUyMS56aXA=",
  "filename_decoded": "\u300aBorland C++ 4.0 for Windows\u9762\u5411\u5bf9\u8c61\u7a0b\u5e8f\u8bbe\u8ba1 OWL2.0 for C++\u300b_12766521.zip",
  "filesize": 69783252,
  "md5": "1d99f6543656bfe5a0a47fb3815702b2",
  "header_md5": "099ae813dcf878e4c5f3ab028737a07b",
  "sha1": "f7bb0c6e234c78cd8241200fb9418c15453fbdec",
  "sha256": "f0cde8ebbd0c99a23b26323fe7f0aec8ffa8030084033c98efe62aec79c917fe",
  "crc32": 4004704634,
  "zip_password": "",
  "uncompressed_size": 74430518,
  "pdg_dir_name": "\u00ed\u2562Borland C++ 4.0 for Windows\u251c\u00b5\u2567\u2265\u2562\u2558\u2567\u2264\u2502\u2560\u2568\u2265\u2554\u03a6\u255d\u255e OWL2.0 for C++\u00ed\u2556_12766521",
  "pdg_main_pages_found": 263,
  "pdg_main_pages_max": 263,
  "total_pages": 269,
  "total_pixels": 1624691232,
  "pdf_generation_missing_pages": false
}